Popular Applications Series

Developing Utilities in Assembly Language

FREE Source Code Disk

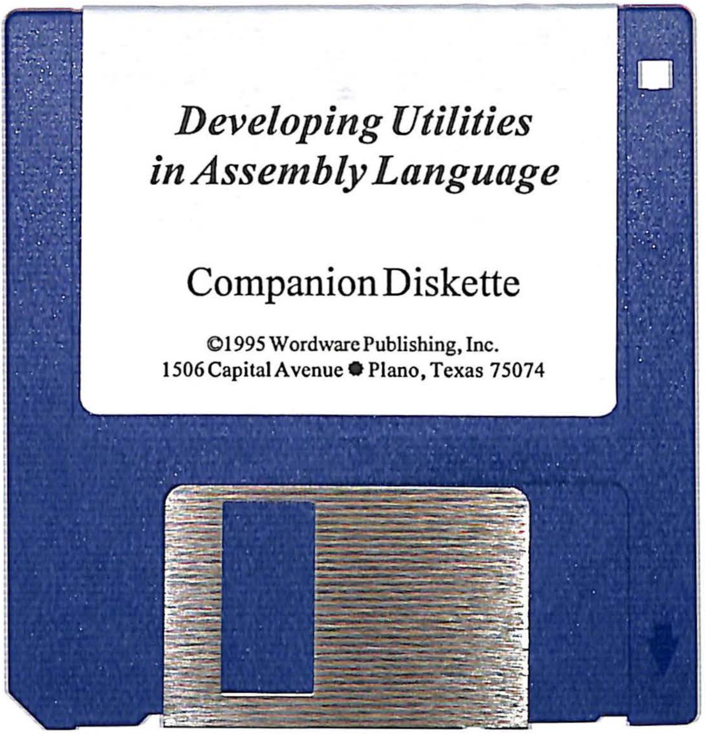
ASSEMBLY LANGUAGE

Learn how to:

- ☑ Design and Complete Assembly Language Programs
- ☑ Create DOS Utilities
- ☑ Enhance Applications with Supplied Source Code
- ☑ Create TSR Utilities
- ☑ Expand Programs from Supplied Source Code

And much, much more!

Deborah L. Cooper



Developing Utilities in Assembly Language

Companion Diskette

©1995 Wordware Publishing, Inc.
1506 Capital Avenue ● Plano, Texas 75074

Developing Utilities in Assembly Language

Deborah L. Cooper

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Cooper, Deborah L.
Developing utilities in assembly language / by Deborah L. Cooper.
p. cm.
Includes index.
ISBN 1-55622-429-X
1. Assembler language (Computer program language). 2. Utilities (Computer programs). I. Title.
QA76.73.A8C66 1994
005.4'2--dc20 94-12424
CIP

1506 Capital Avenue
Plano, Texas 75074

Printed in the United States of America

ISBN1-55622-429-X
10 9 8 7 6 5 4 3 2 1
9411

Limits of Liability and Disclaimer of Warranty

The author and publisher of this book have made every effort to ensure the accuracy of this book's contents. However, neither the author nor the publisher make any warranty of any kind, expressed or implied, with regards to the programs and contents in this book. Neither party shall be liable in any event of incidental or consequential damages in connection with, or arising out of, the use of the programs or material contained herein.

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(214) 423-0090

To the memories of my mother, Vi Cooper, and my grandmother, Elna Bakken. Rest in peace. I love and miss you both.

Acknowledgements

A project such as this would not have been possible were it not for the help of many people.

I'd like to express special thanks to my brother, Mike. His neverending love and support gives me the encouragement to follow my dreams. And, with the birth of his daughter, Michelle, he's made me one proud Auntie!

I'm also greatly indebted to my friend and mentor, Denise Shariatmadari. It is through her wisdom and guidance that my life has become much more rewarding.

I would also like to thank my friend, June Hale, the first person to look at and criticize my manuscript. She taught me some important concepts that helped me tremendously when writing this book.

Finally, I would like to thank all those at Wordware Publishing. Their hard work and expertise have helped to make this project a reality.

Contents

Chapter 1

INTRODUCTION

Walk into any bookstore that features computer programming books for the IBM PC, in particular those related to assembly language programming, and you'll be sadly disappointed. Certainly, you can find several volumes that describe the various components of the 8086-8088 instruction set and how they are used. You will also surely find a plethora of books devoted to descriptions of the hundreds of DOS and BIOS function calls built into the PC's operating system. However, you will find few, if any, books that will show you how to develop a program from scratch.

This book is called "Developing Utilities in Assembly Language," which exactly describes its theme. The key word here is "developing." By studying the source code listings for the programs contained in this book and following the detailed explanations of how the programs were constructed, you will learn a great deal about assembly language programming on the PC—things that the other books neglect to cover.

This book describes many useful programming techniques you can master and use when creating your own programs. You will learn how to design programs from start to finish. To this end, it is hoped that this book will spark your imagination, creativity and productivity. Therefore, don't just give the book a once-over read. Study each section and the techniques and discussion presented. Think about how to undertake some of the challenges presented under "Projects". Take the time to modify the source code files, add new features to the programs, or create entirely new programs from the ideas presented. The more time you take to play and learn, the more productive and rewarding will be your programming hobby or career.

It is assumed that you are familiar with using DOS and BIOS function calls and have a good working knowledge of the PC's instruction set.

You may, however, find it useful to have handy a good reference book or two that contains this information. Although this book is not a beginner's guide to learning assembly language, it will nevertheless be useful to people at all skill levels of assembly language programming, since no other book can show you how to develop programs from start to finish using the PC's resources.

The utilities presented in this book can and should be used as a stepping-stone. That is, you can modify and enhance these programs to add more features, to optimize them or to use as starting points for other utilities.

How This Book Is Organized

Each section of this book discusses the development and theories used to create a working program. To this end, the sections are presented in a uniform manner that is easy to follow. Each section contains the following:

- a description of the program's purpose and instructions on how to use it
- a list of the DOS and BIOS function calls used in the program
- a very detailed explanation of how the program was developed along with special techniques used
- a short summary of what is covered in this section
- one or more sources to consult for additional information
- the actual ASM source code listing for the program itself

Because of the book's organization, it is not imperative that you start reading from page one. To explain, the programs contained in the first half of this book are all regular DOS utilities; the programs contained in the second half are TSR utilities. You may skip around to those sections that you are most interested in right now and then go back to the others later on. If you're a beginner, you will most likely start reading from the beginning of the book and progress to the end, learning about new methods and techniques as your expertise expands and grows.

Each section describes one complete assembly language program. To this end, all the information is together in one section. This makes it far easier to learn as you follow along.

What You Will Need with This Book

All the programs in this book were developed with Turbo Assembler 2.0. Even if you don't use this particular assembler package, the programs can be assembled without any modifications to the source code with another assembler package such as the Microsoft Macro Assembler.

You should have the following equipment configuration (minimum) to use the programs presented:

- IBM PC, XT, AT, or compatible
- 640K random access memory (RAM)
- at least one floppy or hard disk (hard disk recommended)
- a monochrome or color video display
- a dot matrix printer
- DOS 3.3 or greater

Creating the Programs

Each program listing should be typed into your text editor and saved as an ASCII file. Then, compile the program with the following command sequence:

```
TASM filename;
TLINK filename;
EXE2BIN filename filename.com
```

Note that each program is compiled into a COM-style program. This makes the final version of the utility smaller, taking up less disk space, and makes the program quicker to load by the operating system.

Program Disk Available

You might want to save yourself a lot of time typing, since the source code for all eight utilities presented in this book is quite long. A diskette containing the source code for all eight programs is packaged with this book.

Comments and Suggestions

I would like to hear your comments, suggestions for improvements or corrections about any aspect of this book. You may contact me at the address below. All letters will be answered.

Deborah L. Cooper
19901 - 55A Avenue
Langley, B.C.
V3A 3X4

Overview of Programs

This section briefly describes the utilities presented in this book. From this list, you can see that many different types of programs can be written in assembly language.

These types of programs include DOS utilities, TSR (terminate and stay resident) utilities and special "add-on" programs to be used with applications such as WordPerfect. Each section explains how the program was created in assembly language, including special techniques used, as well as enhancements you can attempt on your own.

Here are the programs:

MACLIST.ASM An add-on utility for WordPerfect that displays the description for macro files. The resulting list can be directed to the screen, the printer, or to a newly created WordPerfect document file.

DIRNAME.ASM Renames subdirectories. This operation can normally be done only with the DOS "SHELL" program.

FILEFIND.ASM Quickly finds any file on a floppy or hard disk. This program is a prime example of recursive programming techniques.

TRAPBOOT.ASM Prevents a user from accidentally rebooting the computer system by disabling the keystroke combination Ctrl+Alt+Del.

TRAPDEL.ASM A utility that intercepts DOS's file delete function and places the to-be-deleted file in a special

GARBAGE directory. Demonstrates how DOS functions can be manipulated.

SAFE.ASM Disables the DOS "FORMAT" command. It provides an example of using undocumented DOS function calls and further shows how to add new DOS commands or modify existing ones.

CAPSLOCK.ASM From the DOS command line, this utility produces normal characters, regardless of whether the CapsLock key is engaged or not.

ICU.ASM Displays a highlight bar across the screen where the cursor is located. This program is useful on laptop and notebook computers, as well as standard desktop machines, where you cannot easily see the cursor.

Chapter 2

MACLIST

MACLIST is an add-on utility for WordPerfect that displays the description for macro files. The resulting list can be directed to the screen, the printer, or to a newly created WordPerfect document file.

Enhancing Existing Applications

As you probably know from having used many software packages, there is a great variety of programs available for the PC. These types of programs can be categorized into DOS utilities, stand-alone application programs, device drivers, and terminate and stay resident (TSR) utilities. Programs have also been developed that add new features to existing commercial applications such as WordPerfect Corporation's word processing package, WordPerfect.

WordPerfect enables you to create macro files that perform any number of tasks related to word processing. When you first create a macro, you can enter a description for future reference. Unfortunately, there is no easy way to obtain a list of all the macro files and their corresponding descriptions.

MACLIST, the utility presented here, was developed to solve this problem. MACLIST will compile a list of the names of these macro files and their descriptions. This generated list can then be directed to the screen, to the printer, or to a newly created WordPerfect document file.

Functions Used in MACLIST.ASM

Int 10h, AX=02h	Clear screen
Int 16h, AH=00h	Read character from keyboard
Int 17h, AH=02h	Get printer status
Int 21h, AH=09h	Display string
Int 21h, AH=0Ah	Buffered keyboard input
Int 21h, AH=1Ah	Set disk transfer address
Int 21h, AH=3Bh	Set current directory
Int 21h, AH=3Ch	Create file
Int 21h, AH=3Eh	Close file
Int 21h, AH=3Fh	Read from file or device
Int 21h, AH=40h	Write to file or device
Int 21h, AH=4Ch	Terminate process with return code
Int 21h, AH=4Eh	Find first matching file
Int 21h, AH=4Fh	Find next matching file

How to Use MACLIST

To use MACLIST, type the program's name at the DOS prompt. After you press ENTER, the program will prompt you to enter the name of the directory where you store your WordPerfect macro files. The name you type may include an optional drive letter.

Next, you will be asked where you want the compiled list to be sent—to the printer, the screen, or a WordPerfect document file. Type one of the letters S, P, or D to select the desired option. At this point, you may exit back to DOS by pressing the ESCape key if you wish to terminate the program.

Once your option has been selected, MACLIST will produce the report on the output device or file. When it has completed its work, MACLIST will exit back to DOS.

What Kind of Code Is This?

When you type a command at the DOS prompt, the operating system immediately checks to see if your request was to execute an internal DOS command. An internal DOS command, such as DIR, is stored in COMMAND.COM. COMMAND.COM, of course, resides in memory because it is the mechanism that the machine runs under. If the command you requested was not an internal DOS command, then DOS checks to see if it was a COM or EXE program file. DOS automatically assumes that a file with either one of these two extensions is a program, i.e., a file of executable code that it can load and run. These executable files include utilities, application programs, and DOS's own external commands like CHKDSK or FORMAT.

In addition, there are several differences between COM and EXE program files. A program written in the COM-style is limited in size to a maximum of 64K in length. Since a COM program cannot exceed 65,536 bytes, all of its machine code instructions, data and stack declarations must always reside in one segment of memory.

On the other hand, EXE files are not limited in size whatsoever. A single EXE program file can encompass many memory segments and usually has at least three separate segments for its code, data and stack requirements.

COM programs are loaded by the operating system into memory directly above the Program Segment Prefix (PSP). Since the PSP is 256 bytes long, all COM programs must be originated at offset address 100h. This is why all COM programs begin with the ORG 100h statement. EXE program files are also loaded into memory just above the PSP. However, the order of the segments loaded into memory may vary from program to program since a program's code, data and stack segments could be in a different order than those for a COM program.

How does DOS know if it is executing an EXE or a COM program? It looks at the first two bytes of the file. The file is an EXE file if these first two bytes are "MZ." A COM-style program does not have this identifying signature.

The utility presented in this section is called MACLIST, an example of a COM-style program.

In the Beginning

When MACLIST is executed at the DOS command prompt, it immediately clears the screen by using a ROM BIOS function, as shown:

```
begin:  mov     ax,02h                  ;clear the screen
        int     10h                     ;call bios
```

Function 02h of Int 10h, Set Cursor Position, accepts four parameters: AH holds the function code (02h), BH holds the video page number, and DH and DL hold the row and column positions, respectively.

When DOS loads MACLIST, it knows it is executing a COM program because of the filename's extension. Since COM files only contain binary bytes that tell the computer what to do and are an exact image of these instructions, it knows it must execute the program beginning at offset 100h. When a COM program is first loaded, most of the registers contain a value of zero. This is why the AX register is the only register that has to be implicitly set by MACLIST to clear the screen with an Int 10h function call.

Once the screen is cleared, MACLIST can display its copyright notice using Function 09h of Int 21h, Display String. This DOS function requires that DS:DX hold the address of a string of ASCII characters, all terminated by a final '$' byte. Note that the dollar sign character is not displayed on the screen; it is only used by DOS to signal the end of the string. In addition, the function code (09h) is placed in AH.

```
        mov     dx,offset copywr     ;point to copyright notice
        mov     ah,09h               ;display string function
        int     21h                  ;call dos
```

Getting Input from the Keyboard

Since MACLIST must find out where the WordPerfect macro files are stored, the only way it can determine this is to prompt you for the information. This is done with the following lines of code:

```
askdir: mov     dx,offset wdir       ;which directory prompt
        mov     ah,09h               ;display string function
        int     21h                  ;call dos
        mov     dx,offset mname      ;point to input buffer
        mov     ah,0ah               ;line input function
        int     21h                  ;call dos
```

Function 0Ah of Int 21h, Buffered Keyboard Input, enables a program to accept a line of input via the keyboard. As the text is typed, it is saved in the destination buffer. This DOS service is terminated when it receives a carriage return (0Dh) character. This terminating carriage return byte is also saved in the buffer.

Like other DOS function calls, Function 0Ah requires that DS:DX holds the address of a buffer where the incoming data is to be saved. However, this buffer must be in a specific format. The first byte, which is set by the program, holds a value that represents the maximum number of bytes to be read. After the function has returned, the second byte of this buffer will be set to the actual number of characters read. This count does not include the terminating carriage return byte, though it is indeed stored in the buffer. The actual characters typed on the keyboard are stored consecutively, starting at the third byte of the buffer.

It should be noted that Function 0Ah can also be redirected to look for input from a file or device as well as the keyboard. This could be useful in many situations. You will see how to redirect output to a file or device using Function 40h of Int 21h later in this chapter. This same technique can be used to redirect other functions such as Function 0Ah.

There are two advantages to using Function 0Ah to get input. The first advantage is that should the buffer be completely filled without receiving a carriage return, all subsequent input is ignored and a bell is sounded until the 0Dh byte is received. Knowing this, you can be assured that a user can't go on forever typing, messing up your carefully designed input screen!

The second advantage to using Function 0Ah is that it allows for DOS's type-ahead capability. This means that you can type very fast without losing any characters even if the currently running program is busy doing a disk access or other such work. DOS will continue saving the keystrokes in its type-ahead buffer and the application program can process them when it has the time to do so. In addition, all the normal editing functions (delete, insert, etc.) provided by DOS are available when this service is used.

Where Are We on the Drive?

MACLIST must read the files stored in the WordPerfect macro subdirectory. Therefore, the program asks the user to enter the name of the directory where these macro files reside. If the user just presses the ENTER key without typing a pathname, then the current directory on the default disk drive is used.

Function 47h of Int 21h, Get Current Directory, requires three parameters. The AH register is loaded with the function code (47h). A drive code, where zero specifies the default disk drive, 1=A, etc., is placed in the DL register. In addition, DS:SI must point to a buffer that is 64 bytes long, the longest length possible for a pathname. The following code shows how this is done:

```
        mov     si,offset orig_dir     ;destination buffer
        mov     al,'\'                 ;get a leading slash bar
        mov     [si],al                ;store in buffer
        inc     si                     ;bump buffer pointer
        mov     ah,47h                 ;get current directory function
        xor     dl,dl                  ;for default drive
        int     21h                    ;call dos
```

Notice that SI is loaded with the address of the buffer ORIG_DIR, which, on return from Function 47h, will hold the ASCIIZ pathname. However, since this function call does not write a leading backslash (\) character before storing the pathname, one is inserted in the buffer before the function is called.

An ASCIIZ string is a series of characters that contains the drive, path, filename (including the filename extension), all terminated by a zero (0) byte. Therefore, whenever a DOS function requires a pointer to an ASCIIZ filespec, this is the format you would use. Not all of the parameters need to be specified in the ASCIIZ string. The following are all valid examples:

```
        NAME1    DB     'C:\WP\LETTER.DOC',0
        NAME2    DB     'C:\WP',0
        NAME3    DB     'LETTER.DOC',0
```

Once the pathname has been stored for later use, the program continues with the statements:

```
        mov     bx,offset mname+1      ;BX=actual length of pathname
        mov     al,[bx]                ;get length in AL
        cmp     al,0                   ;was a pathname entered?
        je      which                  ;no, use default directory
        mov     bl,mname+1             ;yes, make the pathname
        mov     bh,0                   ;into an ASCIIZ string
        mov     [mname+bx+2],0         ;ending in a zero byte
```

What this section of code does is convert the pathname, if one was actually entered at the label ASKDIR, into an ASCIIZ string. An ASCIIZ string is a string of characters terminated by a zero (0) byte. Earlier, when Function 47h retrieved the pathname for the default disk, it was stored in the buffer ORIG_DIR as an ASCIIZ string. But, since Function 0Ah terminates its input buffer with a carriage return, the carriage return byte has to be replaced with a zero byte.

An interesting technique should be pointed out about the above lines of source code. If you recall, the input buffer used by Function 0Ah automatically calculates the number of characters that were stored in the buffer. This count is then saved in the second byte of the input buffer. In the code shown above, this value was retrieved from the buffer directly to determine if a pathname was actually entered and this was also used to determine the address of the carriage return byte. The only other way we could have determined the string's length would have been to scan the entire string searching for the 0Dh byte and keeping a tally of the total number of characters we bypassed as we advanced through the buffer.

However, since Function 0Ah calculates the string's length for us, we only need to load BX with the address of the input buffer, increment that address by one to point to the count byte, and then retrieve the count value into the AL register.

One Keystroke at a Time

Once MACLIST knows where to find the macro files, control jumps to the label WHICH, and a prompt is displayed using Function 09h. This prompt asks the user to select which output file or device he wants to direct the compiled list to—the screen, printer, or WordPerfect document file. Then Function 00h of Int 16h is called to read a single character from the keyboard. On return, AH holds the keyboard scan code and AL holds the ASCII character. Once the ASCII character is in AL, it is converted to an uppercase character and then compared with the possible options, as shown here:

```
inkey:  mov    ah,00h         ;get a keystroke
        int    16h            ;call bios
        and    al,5Fh         ;convert to uppercase
        cmp    al,'S'         ;output to screen?
        je     scr            ;yes, fix it
        cmp    al,'P'         ;output to printer?
        je     prt            ;yes, fix it
        cmp    al,'D'         ;output to WP document?
        je     outdoc         ;yes, set it up
        cmp    al,27d         ;was ESCape key pressed?
        je     esc_key        ;yes, process it
        jmp    inkey          ;go back if invalid choice
```

get a keystroke which is put into al

The ASCII character in AL is first converted to uppercase by the statement AND AL,5Fh. This was done for a good reason: it saves on the number of comparisons that need to be made to see if the keystroke is a valid choice. In other words, if the character was not converted, a comparison would have had to be made for both the uppercase and lowercase letters S, P, and D. A test is also made here to see if the ESCape key, represented by ASCII code 27, was pressed to gracefully exit back to DOS if the user does not want to continue using the MACLIST program at this time. All programs should have a graceful method of exiting back to DOS.

Converting Characters to Uppercase and Lowercase

You can use a very neat technique when you need to convert a character to uppercase. If you look at Appendix B, a chart of the ASCII character set, you'll find the binary representation for all 256 possible characters, numbers and other special characters available on the IBM PC. By comparing a character such as the letter "A" to its lowercase equivalent, you'll notice that bit 5 is reset (off) for uppercase characters but it is set (on) for lowercase characters. However, also note that this only applies to letters, not the numbers or other special characters.

To put it simply, all you have to do to convert a character to uppercase is turn bit 5 off. Conversely, to convert the character to lowercase, you would turn bit 5 on. The logical AND instruction with a value of 5Fh will turn bit 5 off, thereby producing an uppercase character. Similarly, using the OR instruction with a value of 20h, will convert the character to lowercase.

WordPerfect Document Formats

As with many commercial applications available on the market today, WordPerfect creates its documents in a proprietary format. Having the ability to write files in WordPerfect's native format is a neat feature and a bonus included in the MACLIST program. The whole idea behind this program is to provide a method of viewing descriptions attached to macro files. It makes sense, then, that this program includes the code necessary to create a file that is readable by WordPerfect. When MACLIST has completed its work, you can retrieve the document containing the macro descriptions into the word processor for further processing, if desired.

A WordPerfect document is simply a file consisting of two parts. The first part, which is 76 bytes long, identifies the file as having been created by WordPerfect. It is made up of a 16-byte file prefix and a 60-byte block of data containing information relating to the graphic images used in the document, the selected printer, and other such items needed by WordPerfect to produce the document.

The second part of a WordPerfect document is made up of the text, complete with formatting codes such as indent, page breaks, and so on.

From this, its easy to see that MACLIST must begin by creating a file on the disk. The routine starting at the label OUTDOC displays a prompt and uses Function 0Ah of Int 21h to allow you to enter a filename for the WordPerfect document that is going to be created. As explained earlier in this section, the filename is converted to an ASCIIZ string.

```
outdoc: mov     dx,offset wfile         ;which filename prompt
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     dx,offset wname         ;point to input buffer
        mov     ah,0ah                  ;line input function
        int     21h                     ;call dos
        mov     bx,offset wname+1       ;BX=# of bytes entered by user
        mov     al,[bx]                 ;get length in AL
        cmp     al,0                    ;was a filename entered?
        je      out_b                   ;no, exit with error then
        mov     bl,wname+1              ;yes, make the filename
        xor     bh,bh                   ;into an ASCIIZ string
        mov     [wname+bx+2],0          ;ending in a zero byte
```

Once a filename has been entered and converted into ASCIIZ format, the program jumps to the label OUT_A, as shown below:

```
out_a:  mov     dx,offset wname+2       ;DX=filespec to create
        mov     ah,3ch                  ;create new file function
        mov     cx,00h                  ;normal file attribute
        int     21h                     ;call dos
        jnc     out_1                   ;go if no errors
```

Function 3Ch of Int 21h, Create File, is used to open and create a new file on the disk. On entry, the AH register is loaded with the function code (3Ch), CX is set to the attribute to be used when creating the file (to be discussed shortly) and DX holds the address of the filename to be given to the newly created file. Note that if the filename does not contain a drive and/or pathname, it will be created in the current default directory on the currently selected disk drive.

On return from the Create File function, a handle will be placed in the AX register if the file was successfully created; otherwise the Carry Flag will be set and an error code will be returned in AX. If a file by the same name already exists, an error condition will not be generated. Instead, the file's contents are erased because its length is truncated to zero by Function 3Ch.

However, an error would occur if the existing file's attribute was set to read-only, if any part of the pathname addressed by DX did not exist or if an invalid file attribute was specified in CX.

It is important that a program preserve the handle number returned in the AX register. Other DOS file functions that manipulate this particular file will need to use this handle number to identify the correct file on the disk.

File Handles

Whenever DOS is asked to open an existing file or create a new file, it assigns a unique 16-bit number to the file. This number is called a file handle.

This method of accessing files was developed with DOS version 2.0 to simplify access to files and devices. In previous versions of DOS, you were forced to use a File Control Block (FCB) structure in order to read or write data to a file. Using file handles eliminates a lot of the preparation work required for FCB file access. Instead, you can access a disk file or device by simply specifying a file handle number. You don't have to bother setting up separate data structures as you must when using FCBs.

Therefore, when we want to write data to a file, we can use Function 40h of Int 21h, Write to File or Device. This service call requires that BX holds a file handle number. When data is written to the file, DOS uses the file handle number, not an ASCII filename. In addition, Function 40h can be used to send data to a device such as a dot matrix printer.

DOS uses the first five file handles, numbered 0 through 4, as shown in the table below. These first five handles are always open and ready to use. They provide DOS with the ability to redirect input and output, depending on how the file handles are manipulated. Any additional files opened or created by application programs are numbered starting with 5.

0	Standard Input Device (keyboard)
1	Standard Output Device (screen)
2	Standard Error Device (screen)
3	Standard Auxiliary Device
4	Standard Printer Device (PRN or LPT)

MACLIST uses DOS's I/O redirection capability to output its records to the screen, the printer or to a WordPerfect document. Earlier, the program asked which output option was to be used. No matter which option is selected, its file handle is saved in the variable OUTPUT. If the option is a WordPerfect document, a disk file is opened with Function 3Ch and this file's handle is saved in OUTPUT. If the output option selected was the printer or the screen, then MACLIST sets the handle to a value of 4 or 1, respectively.

Writing to a File

Once the handle is saved in the variable OUTPUT, the next section of code writes a 76-byte block of data to the file. As was explained earlier in this section, this block of data contains the first section required by all documents that are in WordPerfect's proprietary format.

To write data to a file, Function 40h of Int 21h is called with DX pointing to the buffer of data to be written to the file. A count of the number of bytes in this buffer is put in CX, and BX holds the handle number.

```
mov     dx,offset header      ;data to write to file
mov     bx,output             ;file number
mov     ah,40h                ;write to file function
mov     cx,76d                ;this many bytes to write
int     21h                   ;call dos
```

to a file

If no errors occur when MACLIST attempts to write the header information to the file, the program branches to the label FIND> In the event that MACLIST was unable to write data to the file, the program exits back to DOS with an error message.

Changing Directories

Before MACLIST can begin searching the disk for macro files, a little preparation work needs to be done. First, MACLIST must change to the directory specified in the MNAME buffer, that is, the directory on the disk where WordPerfect stores its macro files. To do this, we use Function 3Bh of Int 21h, Set Current Directory.

```
find:   mov     dx,offset manem+2      ;new directory name
        mov     ah,3bh                 ;change directory function
        int     21h                    ;call dos
        mov     count,0                ;set record counter to zero
        mov     bx,offset mac_rec      ;BX=points to record buffer
        mov     place,bx               ;save starting address for it
        mov     dx,offset dta          ;point DX to disk transfer buffer
        mov     ah,1ah                 ;set DTA function
        int     21h                    ;call dos
```

Function 3Bh requires only two parameters. The function code (3Bh) must be placed in AH and the name of the directory you want to switch to is addressed by DX. Once this function call is executed, this subdirectory becomes the new default directory.

Next, the Disk Transfer Address is set to MACLIST's own buffer called, appropriately enough, DTA.

The Disk Transfer Address

The Disk Transfer Address, or DTA as it is referred to, is an area of memory set aside by DOS or application programs. This area of memory is used by many DOS functions as an I/O buffer. For example, the Find First Matching File and Find Next Matching File functions call store information relating to the found file in the DTA.

It is especially important to use a new DTA buffer in your own programs. The default DTA set by DOS is also the same area of memory used to store the command line parameters, i.e., the PSP. Therefore, care must be taken to save the contents of the command line before performing any disk operations, including setting a new location for the DTA via Function 1Ah of Int 21h. If this step is not taken, then the very next disk operation will overwrite the contents of the PSP's command line parameters, if any are stored there.

The size of the memory buffer you allocate for the DTA can be any length you desire. However, it is important that you calculate this size appropriately. For example, if your program needs to read data from a file stored on disk, and that block of data is 2,000 bytes long, then your DTA must be able to accommodate this size. If your DTA buffer is only 1,000 bytes long, then you would not only lose half of your data, but that same data could very possibly overwrite a portion of your program's code or other data. Therefore, you could spend a huge amount of time trying to debug your program when in fact you just need to increase the size of your DTA buffer!

Locating Directories and Files

Since MACLIST needs to find the macro files stored in the designated subdirectory, it relies on two DOS functions tailored to this purpose. To begin searching for a file, DOS Function 4Eh is used. This function finds the first occurrence of the file specification. To do this, the function code (4Eh) is loaded into AH with the attribute of the files to be searched for loaded into CX. Next, DS:DX points to an ASCIIZ (a string terminated by a zero byte) that contains the name of the file to be looked for. The ASCIIZ filename may contain the wild card characters '*' and '?'. If wild card characters are specified, then only the first occurrence of a matching filename will be returned by Function 4Eh. Subsequent calls to Function 4Fh will retrieve the next matching wildcard file.

When specifying the file attribute the function is to look for, CX must be loaded with one or a combination of the values shown in the following table. The operating system uses a file's attribute to determine what actions can be taken on the file. For example, if you were to try to open a file with an attribute of 01h, the read only attribute, and you then attempted to write data to the file, DOS will interrupt you with an error message.

Any combination of these bits can be used with one exception. If any of the bits in CX indicate that a Volume Label file is to be searched for, only Volume Label files will be returned. A disk may have, at most, one Volume Label file, which is used to give the disk a unique name and is stored only in the root directory. Any other combination of attributes will return all normal files as well as those hidden, system and/or directory files the function finds on the disk that match the file specification. In addition, if the file has an attribute value of 20h, then this marks the file as having been modified or created since the disk was last backed up. When CX is loaded with a zero value, only normal files are found.

DOS File Attributes for File Find Functions

00h	Normal	08h	Volume Label
01h	Read Only	10h	Subdirectory
02h	Hidden	20h	Archive
04h	System	40h	Unused

If Function 4Eh was unable to find at least one matching file, then the Carry Flag will be set, indicating an error condition which is returned in AX.

On the other hand, if Function 4Eh was able to find a file, the DTA will contain information about the found file. To this end, you must remember to use Function 1Ah of Int 21h to set the DTA to a buffer in memory reserved for this purpose before initiating a search call. The buffer you set aside for the DTA when using the Find File function calls should be 43 bytes long. When the function has found a directory entry that matches the target specification, it fills the DTA buffer with information about this one individual file, as depicted in the table below.

The Disk Transfer Address Buffer

0	Drive code (1=A, 2=B, etc.)
1-12	Filename, padded with spaces, no period separating name and extension, wild card characters '*' and '?' are replaced
14-15	Position of filename in directory. The first directory is 0, the second 1, etc. Erased file and volume positions are included in this count.
16-17	Directory (path) position
18-20	Reserved by DOS
21	Attribute of file
22-23	Time file was created or last modified
24-25	Date file was created or last modified
26-29	Size of file in bytes. Low word, followed by high word
30-43	Filename in ASCIIZ format. Filename is not padded with spaces, a period separates the filename and the extension, wild card characters '*' and '?' are replaced

As soon as one of the Find File functions locates a file in the macro directory, the program branches to the label OPEN. Here, a different DOS function attempts to open the file just found. Function 3Dh requires that DX hold the address of the ASCIIZ filename to be opened. In this case, the filename was put into the DTA buffer by the Find File function. Therefore, we point DX to this buffer and then add the offset of 30 bytes. This will make certain DX points to the first byte of the actual filename.

Since we only want to read the data stored in the file to be certain it is a WordPerfect macro file, we set AL to zero to tell Function 3Dh to open this file for read access only.

If the macro file was opened successfully, Function 3Fh, Read From File, is used to read the first 200 bytes, specified in CX, and stores this data in a buffer called DATABF. The file is then closed using Function 3Eh of Int 21h and the procedure SAVE is called.

The SAVE procedure writes a record to the buffer called MAC_REC. For each filename found in the WordPerfect macro directory, a record is created in the MAC_REC buffer. Each record consists of the DOS filename and its corresponding description. Later on, the records in this buffer will be sorted alphabetically by filename prior to being sent to the output destination.

Although the source code for the SAVE procedure is quite long, it really isn't as complicated as it first appears.

The first section of this procedure reads the filename just found by the Find File function and saves it in the MAC_REC buffer. As stated earlier, the Find File function writes information about each directory entry into the DTA. Therefore, the following code retrieves the filename from the DTA, saving it in the MAC_REC buffer:

```
save    proc    near
        inc     count             ;add one to record counter
        mov     di,place          ;DI=destination address
        mov     si,offset dta     ;SI=disk transfer buffer
        add     si,30d            ;move up to ASCIIZ filename
        mov     cx,13d            ;length of filename maximum
save_1: mov     al,[si]           ;get one character of filename
        cmp     al,0              ;end of filename reached?
        je      save_2            ;yes, pad with spaces if short
        mov     [di],al           ;save character in mac_rec
        inc     si                ;bump both buffer
        inc     di                ;pointers
        loop    save_1            ;loop until filename copied
        jmp     save_3            ;and continue
```

```
save_2: mov     al,20h                  ;get a space character
        mov     [di],al                 ;store in mac_rec
        inc     di                      ;bump buffer pointer
        loop    save_2                  ;until it's padded with spaces
```

The first step taken in the above section of code is to increment the record counter by one. The variable COUNT is used to keep track of how many records are placed in the MAC_REC buffer. This value will be used later on in the program when the records are sorted and also when the records are sent to the output file or device.

In addition, the variable PLACE is used to keep track of the last address written to in the MAC_REC buffer. This variable gets updated in the SAVE procedure so MACLIST knows exactly where to store the next record to be written to the MAC_REC buffer stored in memory.

The rest of the SAVE procedure simply copies each byte of the filename from the DTA buffer to the MAC_REC buffer. Since a filename can be up to 12 characters long, SAVE pads those filenames less than this length with space characters. By doing this, each record is uniformly formatted, and both the sort and output routines are easier to write if each record is the exact same length, as will be discussed shortly.

At the label SAVE_3, five blank spaces are also written to the MAC_REC buffer. This in effect creates two columns of information to clearly delineate the fields in every record (the filename, followed by five spaces, followed by the 40-byte macro description). Then, at the label SEE, the macro file's description is retrieved.

The description for the macro file was, if you will recall, read directly from the WordPerfect macro file earlier at the label OPEN. Now all that needs to be done is to read this information from the buffer DATABF. This is done in the exact same way the filename was stored, except each WordPerfect macro file can have a description appended to it that may be up to 40 bytes in length.

When the procedure SAVE issues its RET instruction, control reverts back to the label NEXT, where the Find Next Matching File function processes the next filename in the macro directory. The SAVE procedure is then repeated for each macro file until no more directory entries are found in this directory.

Sorting a File in Memory

Before the records stored in the MAC_REC buffer are output to a file or device, they are sorted alphabetically by filename. The procedure SORT does this work for us.

The SI register holds the address of the MAC_REC buffer, DX is set to the length of each record, and CX holds a count of the total number of records to be sorted, as shown here:

```
sort    proc    near
        mov     si,offset mac_rec      ;records to be sorted
        mov     di,si                  ;into DI as well
        mov     dx,57d                 ;length of one record
        add     di,dx                  ;DI=set to second record
        mov     cx,count               ;number of records in CX
        dec     cx                     ;subtract one to begin with
```

Notice that the number of records in the buffer is decremented by one to start with. This prevents an endless loop from occurring in cases where there is only one record stored in the MAC_REC buffer. In addition, SI and DI are initially set to the starting address of the MAC_REC buffer. Next, the record length is added as an offset to DI. This makes SI point to the first record and DI to the second record.

The code starting at the label SORT_1 saves the record count and the SI and DI buffer pointers on the stack. This is done in case the records are swapped later on and we can find where we left off in the buffer when we need to come back and sort the next set of records.

Next, the REP CMPSB instruction is used to compare the two records addressed by SI and DI. If the two records are already in alphabetical order, the program branches to the label SORT_3, which sets SI and DI to point to the next two records in the buffer that are to be sorted.

However, if the records are not in order already, the following code fragment sorts them:

```
        mov     cx,dx                  ;prepare to compare two records
        push    si                     ;save SI
        push    di                     ;and DI
        rep cmpsb                      ;do the compare now
        pop     di                     ;restore DI
        pop     si                     ;and SI
        jbe     sort_3                 ;continue if already in sort order
        mov     cx,dx                  ;else swap the two records
        push    ax                     ;in the buffer
        push    bx                     ;save
```

```
         push    cx                     ;everything
         pushf                          ;including flags
         mov     bx,0                   ;zero BX to start
sort_4:  mov     al,[si+bx]             ;get source byte
         xchg    al,[di+bx]             ;put in second place
         xchg    al,[si+bx]             ;put in first place
         inc     bx                     ;bump buffer pointer
         loop    sort_4                 ;loop until records swapped
```

This section of the program actually reads one byte from the record pointed to by SI, saving the byte in the AL register. Then the XCHG instruction is used to swap the two characters—in this case, the value in AL with the value in DI+BX and then SI+BX. The BX register is then incremented in preparation for the next two bytes to be swapped. The LOOP instruction tells the computer system to repeat this process for the entire length of the record (remember, CX is the record length temporarily).

Once the two records have been swapped, control passes to the label SORT_3 to continue processing all the records in the MAC_REC buffer. The SORT procedure ends when the record count equals zero.

Sending Data to a File or Device

After the records in the MAC_REC buffer have been sorted alphabetically by filename, the program branches to the procedure WRITE. This procedure is the actual code that reads each record from the MAC_REC buffer one at a time and then sends that record to the screen, printer, or newly created WordPerfect document.

First, MACLIST needs to send a carriage return and a linefeed byte to the device. This is done with the following code:

```
         mov     al,0dh                 ;get a carriage return
         mov     [one],al               ;save c/r in buffer first
         mov     ah,40h                 ;write to file/device function
         mov     bx,output              ;to SCREEN or PRINTER device
         mov     cx,1                   ;write one byte
         mov     dx,offset one          ;buffer of data to write
         int     21h                    ;call dos
         mov     ah,40h                 ;write to file/device function
         mov     al,0ah                 ;get a linefeed character
         mov     [one],al               ;save linefeed in buffer first
         mov     cx,1                   ;write one byte
         mov     dx,offset one          ;buffer of data to write
         int     21h                    ;call dos
```

The WRITE procedure takes advantage of DOS's redirection capabilities. Since MACLIST offers you three different forms of output, it makes sense to use the redirection facility provided by Function 40h of Int 21h, Write to File or Device.

Function 40h expects four parameters on entry: AH must hold the function code (40h), BX must hold the file handle number, CX must contain a count of the number of bytes to be written to the file or device, and DS:DX must hold the address of a data buffer. Since we will be sending data to the device one character at a time, the buffer pointed to by DS:DX is also one byte long.

Once the initial carriage return and linefeed characters have been sent to the output file or device, the program branches to the label WRITE_1, as shown here:

```
        mov     bx,offset mac_rec     ;point to records in buffer
write_1: mov    cx,57d                ;length of one record in CX
write_2: mov    al,[bx]               ;get the character in DL
        mov     [one],al              ;save byte in buffer
        push    bx                    ;save BX
        push    cx                    ;save CX
        mov     ah,40h                ;write to file/device function
        mov     bx,output             ;to SCREEN or PRINTER device
        mov     cx,1                  ;write one byte
        mov     dx,offset one         ;buffer of data to write
        int     21h                   ;call dos
        pop     cx                    ;restore CX
        pop     bx                    ;restore BX
        inc     bx                    ;bump buffer pointer
        loop    write_2               ;loop until one record printed
```

ah, 40h

This simple loop routine sends each record, which is 57 bytes long, to the output file or device. Each byte of the record is sent one at a time; you could easily enhance this routine to output the entire record at one time if you enlarge the ONE buffer to accommodate the size increase, and modify CX to hold a count of the number of bytes to be written to the file accordingly.

A carriage return and linefeed are again sent to the file or device in preparation for the next record's data. The total number of records stored in the MAC_REC buffer was previously stored in the variable COUNT. The WRITE procedure subtracts one from this COUNT value each time a record is processed. When the value in COUNT equals zero, the file is closed using Function 3Eh of Int 21h and the program branches back to the label EXIT.

The routine at the label EXIT restores DOS to the original directory it was in when MACLIST was first executed. Function 3Bh of Int 21h, Set Current Directory, requires that the function code (3Bh) be in AH and that DS:DX holds the segment and offset address of an ASCIIZ pathname. When the function is executed, DOS will be returned to the directory named in the buffer ORIG_DIR, and the program exits back to DOS.

Testing Hardware Devices

Whenever an output device is selected, such as a modem or printer, that device should be checked to make certain it is ready to receive data. To this end, the following code fragment checks the status of the printer.

```
prt:    mov     output,4            ;write to standard device
        mov     ah,02h              ;get printer status
        mov     dx,0                ;for default printer
        int     17h                 ;call bios
        cmp     ah,00010000b        ;is printer on-line?
        je      prt_1               ;no, display error
        jmp     find                ;yes, just continue
```

If you will recall, the variable OUTPUT holds the handle number for the file or device that the macro records will be sent to. Here, we are redirecting the output to the printer by storing the printer's file handle in the variable OUTPUT.

To make sure the printer is ready to operate properly, Function 02h of Int 17h, Get Printer Status, is called. This service requires that AH holds the function code (02h) and that DX holds the printer number. The printer number is set to zero for LPT1, 1 for LPT2 or 2 for LPT3. On return from this service, the AH register will contain the status of the printer.

The table below shows the settings of each bit that is returned in AH. In our routine above, bit 4 is set if the printer is selected and ready to accept data. If this particular bit is zero, then the program branches to the label PRT_1 to display an error message and wait for the printer to be fixed.

0	Printer timed-out
1	Unused
2	Unused
3	I/O error

4	Printer selected
5	Out of paper
6	Printer acknowledge
7	Printer not busy

Summary

MACLIST is a stand-alone DOS utility for use with WordPerfect. It shows how to interface your own utilities with commercially available software packages. Many other such programs exist and are constantly being developed by third-party programmers. MACLIST demonstrates how to:

- create a WordPerfect document the same way the original application creates a document—by creating a special header section at the beginning of the file
- sort a file stored in memory
- read directory entries
- open, close and read data from and write data to files
- redirect output to the screen, printer or disk file
- determine the printer's status

Projects

1. It would be nice if MACLIST were available from within WordPerfect itself instead of as a separate DOS program. Modify MACLIST to make it a pop-up TSR utility.
2. Before the records are output to the file or device, add a header line that includes the date and time the file was created and the pathname where the macro files are stored on the disk. Do this for each page of records that is set to the resulting file.

For Further Study

If you wish to write utilities for use with WordPerfect Corporation's application programs, consult their publication "Developer's Toolkit for

PC Products." Most other software companies publish technical information about their applications as well.

In addition, *PC Magazine* published a program called FILECTRL (July 1991, Volume 10, Number 13). This article describes other word processor document formats, including those for WordPerfect and Microsoft Word.

Program Listing

```
;MACLIST.ASM
;Version 1.2
;<c> 1992 by Deborah L. Cooper
;
;This utility outputs the descriptions for macro files created
;by WordPerfect
;
codesg  segment                         ;start of code segment
        assume cs:codesg                ;set up CS
        assume ds:codesg                ;and DS segments
        org     100h                    ;COM style program
start:  jmp     begin                   ;skip over data area

copywr  db      '═════════════════════════════════════════════'
        db      0dh,0ah
        db      '═   MACLIST - The WordPerfect Macro Utility ═'
        db      0dh,0ah
        db      '═         <c> 1992 by Deborah L. Cooper     ═'
        db      0dh,0ah
        db      '═════════════════════════════════════════════'
        db      0dh,0ah,0dh,0ah,'$'

header  db      0ffh,57h,50h,43h,4ch,00h,00h,00h
        db      01h,0ah,00h,00h,00h,00h,00h,00h
        db      0fbh,0ffh,05h,00h,32h,00h,00h,00h
        db      00h,00h,06h,00h,08h,00h,00h,00h
        db      42h,00h,00h,00h,08h,00h,02h,00h
        db      00h,00h,4ah,00h,00h,00h,00h,00h
        db      00h,00h,00h,00h,00h,00h,00h,00h
        db      00h,00h,00h,00h,00h,00h,00h,00h
        db      00h,00h,08h,00h,7ch,00h,78h,00h
        db      00h,00h,00h,00h

one     db      2 dup(0)                ;temporary data buffer
output  dw      1                       ;1=screen 4=printer output
dta     db      128 dup(0)              ;disk transfer buffer
handle  dw      0                       ;file number
databf  db      200 dup(0)              ;buffer for macro file's data
count   dw      0                       ;number of records in mac_rec
place   dw      0                       ;position within mac_rec
orig_dir db     68 dup(?)               ;current default directory
```

data area

$

```
dofile  db      '*.WPM',0               ;files to look for

nmsg    db      'No WordPerfect macro files exist in this directory'
        db      0dh,0ah,'$'
msg2    db      'Unable to create WordPerfect document','$'
msg3    db      'Unable to write HEADER to WordPerfect document','$'
msg     db      'No description for this Macro is available','$'
o_err   db      'Cannot open file','$'
nosuch  db      'No description available              ',0
sel     db      0dh,0ah,0dh,0ah
        db      'Output to <D>ocument, <S>creen or <P>rinter -> ','$'
noname  db      'You must specify a filename!','$'
wdir    db      0dh,0ah,'Enter name of directory for .WPM macro'
        db      0dh,0ah
        db      'files - <ENTER> to use default directory','$'
mname   db      68                      ;maximum pathname length
        db      ?                       ;actual length in bytes
        db      69 dup(?)               ;pathname entered by user
wfile   db      0dh,0ah
        db      'Enter name for WordPerfect document to create ','$'
wname   db      12                      ;maximum filename length
        db      ?                       ;actual length in bytes
        db      13 dup(?)               ;filename entered by user
prtm    db      0dh,0ah,'ERROR - The printer is not ready!'
        db      0dh,0ah,'Fix the printer and press <ENTER> to continue'
        db      '$'
mac_rec db      22800 dup(0)            ;holds filenames & descriptions

begin:  mov     ax,02h                  ;clear the screen
        int     10h                     ;call bios
        mov     dx,offset copywr        ;point to copyright notice
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
;=============================================================
;Allow the user to specify the directory where he has stored his
;.WPM macro files. If the user presses ENTER in response to the
;prompt, then use the default directory
;=============================================================
askdir: mov     dx,offset wdir          ;which directory prompt
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     dx,offset mname         ;point to input buffer
        mov     ah,0ah                  ;line input function
        int     21h                     ;call dos
;Save the current directory for later
        mov     si,offset orig_dir      ;destination buffer
        mov     al,'\'                  ;get a leading slash bar
        mov     [si],al                 ;store in buffer
        inc     si                      ;bump buffer pointer
        mov     ah,47h                  ;get current directory function
        xor     dl,dl                   ;for default drive
        int     21h                     ;call dos
        mov     bx,offset mname+1       ;BX=actual length of pathname
        mov     al,[bx]                 ;get length in AL
        cmp     al,0                    ;was a pathname entered?
        je      which                   ;no, use default directory then
        mov     bl,mname+1              ;yes, make the pathname
        mov     bh,0                    ;into an ASCIIZ string
        mov     [mname+bx+2],0          ;ending in a zero byte
which:  mov     dx,offset sel           ;output to message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
inkey:  mov     ah,00h                  ;get a keystroke
        int     16h                     ;call bios
        and     al,5fh                  ;convert to uppercase
        cmp     al,'S'                  ;output to screen?
        je      scr                     ;yes, fix it
        cmp     al,'P'                  ;output to printer?
        je      prt                     ;yes, fix it
        cmp     al,'D'                  ;output to WP document?
        je      outdoc                  ;yes, set it up
        cmp     al,27d                  ;was ESCape key pressed?
        je      esc_key                 ;yes, process it
        jmp     inkey                   ;go back if invalid choice
esc_key:jmp     exit                    ;exit to DOS now
scr:    mov     output,1                ;write to standard output (screen)
        jmp     find                    ;and continue
;=========================================================================
;Since we will be creating and writing to a WordPerfect document,
;ask the user for the name of the document to create.
;=========================================================================
outdoc: mov     dx,offset wfile         ;which filename prompt
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     dx,offset wname         ;point to input buffer
        mov     ah,0ah                  ;line input function
        int     21h                     ;call dos
        mov     bx,offset wname+1       ;BX=# of bytes entered bv user
        mov     al,[bx]                 ;get length in AL
        cmp     al,0                    ;was a filename entered?
        je      out_b                   ;no, exit with error then
        mov     bl,wname+1              ;yes, make the filename
        xor     bh,bh                   ;into an ASCIIZ string
        mov     [wname+bx+2],0          ;ending in a zero byte
        jmp     out_a                   ;and continue
out_b:  mov     dx,offset noname        ;no filename message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        jmp     exit                    ;and quit
out_a:  mov     dx,offset wname+2       ;DX=filespec to create
        mov     ah,3ch                  ;create new file function
        mov     cx,00h                  ;normal file attribute
        int     21h                     ;call dos
        jnc     out_1                   ;go if no errors
        mov     dx,offset msg2          ;error opening file
error:  mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
```

— Key Sticky

```
        int     21h                    ;call dos
out_1:  mov     output,ax              ;save file handle first
;============================================================
;Write the HEADER to the WordPerfect document file
;============================================================
        mov     dx,offset header       ;data to write to file
        mov     bx,output              ;file number
        mov     ah,40h                 ;write to file function
        mov     cx,76d                 ;this many bytes to write
        int     21h                    ;call dos
        jnc     out_2                  ;go if no errors
        mov     dx,offset msg3         ;unable to write header
        jmp     error                  ;and continue
out_2:  jmp     find                   ;and continue with rest of routine
;If the user selected to output to the printer, make sure
;the printer is online and ready
prt:    mov     output,4               ;write to standard device (printer)
        mov     ah,02h                 ;get printer status
        mov     dx,0                   ;for default printer
        int     17h                    ;call bios
        cmp     ah,00010000b           ;is printer on-line?
        je      prt_1                  ;no, display error
        jmp     find                   ;yes, just continue
;Printer is not ready. Wait until it is.
prt_1:  mov     dx,offset prtm         ;printer error message
        mov     ah,09h                 ;display string function
        int     21h                    ;clal dos
prt_2:  mov     ah,00h                 ;wait for keystroke
        int     16h                    ;call bios
        cmp     al,0dh                 ;was the ENTER key pressed?
        jne     prt_2                  ;no, wait for it then
;Change to the new directory, if one was specified
find:   mov     dx,offset mname+2      ;new directory name
        mov     ah,3bh                 ;change directory function
        int     21h                    ;call dos
        mov     count,0                ;set record counter to zero
        mov     bx,offset mac_rec      ; BX=points to record buffer
        mov     place,bx               ;save starting address for it
        mov     dx,offset dta          ;point DX to disk transfer buffer
        mov     ah,1ah                 ;set DTA function
        int     21h                    ;call dos
first:  mov     dx,offset dofile       ;DX=file to look for (wildcard)
        mov     cx,00h                 ;files with normal attributes
        mov     ah,4eh                 ;find first matching file
        int     21h                    ;call dos
        jnc     open                   ;go if we found a macro file
        mov     dx,offset nmsg         ;else point to error message
        mov     ah,09h                 ;display string function
        int     21h                    ;call dos
        mov     ah,4ch
        int     21h
```

dx

```
;============================================================
;We have found a macro file in the directory so go print its
;description
;============================================================
open:   mov     dx,offset dta           ;point to DTA data
        add     dx,30d                  ;move up to ASCIIZ filename
        mov     ah,3dh                  ;open file function
        mov     al,0                    ;read access only
        int     21h                     ;call dos
        jnc     read                    ;go if opened successfully
        mov     dx,offset o_err         ;else display error message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        jmp     exit                    ;and exit to DOS
read:   mov     handle,ax               ;save file number
        mov     dx,offset databf        ;point to buffer
        mov     ah,3fh                  ;read from file function
        mov     bx,handle               ;from this file
        mov     cx,200d                 ;maximum number of bytes to read
        int     21h                     ;call dos
        mov     ah,3eh                  ;close file function
        mov     bx,handle               ;file number in BX
        int     21h                     ;call dos
;============================================================
;We have a description, so send filename and description to
;the mac_rec buffer now
;============================================================
        call    save                    ;save record to mac_rec buffer
;============================================================
;Go process the next macro file in this directory, if there
;is one
;============================================================
next:   mov     dx,offset dofile        ;DX=points to wildcard filename
        mov     ah,4fh                  ;find next matching file
        int     21h                     ;call dos
        jc      no_more                 ;no more files found, quit
        jmp     open                    ;yes, go process it then
;============================================================
;Now that all the records are saved in mac_rec buffer, we
;need to sort them by filename in alpha order
;============================================================
no_more:call    sort                    ;sort the records
        call    write                   ;send output to screen or printer
exit:   mov     dx,offset orig_dir      ;original default directory
        mov     ah,3bh                  ;change directory function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
exit2:  mov     dx,offset msg           ;no description message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        jmp     exit                    ;and quit
```

```
;==============================================================
;WRITE sends the records in MAC_REC either to the printer
;or to the screen
;==============================================================
write   proc    near
;First we need to send a carriage return and linefeed to
;the selected output device
        mov     al,0dh                  ;get a carriage return
        mov     [one],al                ;save c/r in buffer first
        mov     ah,40h                  ;write to file/device function
        mov     bx,output               ;to SCREEN or PRINTER device
        mov     cx,1                    ;write one byte
        mov     dx,offset one           ;buffer of data to write
        int     21h                     ;call dos
        mov     ah,40h                  ;write to file/device function
        mov     al,0ah                  ;get a linefeed character
        mov     [one],al                ;save linefeed in buffer first
        mov     cx,1                    ;write one byte
        mov     dx,offset one           ;buffer of data to write
        int     21h                     ;call dos
        mov     bx,offset mac_rec       ;point to records in buffer
write_1:mov     cx,57d                  ;length of one record in CX
write_2:mov     al,[bx]                 ;get the character in DL
        mov     [one],al                ;save byte to be output in buffer
        push    bx                      ;save BX
        push    cx                      ;save CX
        mov     ah,40h                  ;write to file/device function
        mov     bx,output               ;to SCREEN or PRINTER device
        mov     cx,1                    ;write one byte
        mov     dx,offset one           ;buffer of data to write
        int     21h                     ;call dos
        pop     cx                      ;restore CX
        pop     bx                      ;restore BX
        inc     bx                      ;bump buffer pointer
        loop    write_2                 ;loop until one record printed
;send c/r and linefeed now
        push    bx                      ;save buffer pointer
        mov     al,0dh                  ;get a carriage return
        mov     [one],al                ;save c/r in buffer first
        mov     ah,40h                  ;write to file/device function
        mov     bx,output               ;to SCREEN or PRINTER device
        mov     cx,1                    ;write one byte
        mov     dx,offset one           ;buffer of data to write
        int     21h                     ;call dos
        mov     ah,40h                  ;write to file/device function
        mov     al,0ah                  ;get a linefeed character
        mov     [one],al                ;save linefeed in buffer first
        mov     cx,1                    ;write one byte
        mov     dx,offset one           ;buffer of data to write
        int     21h                     ;call dos
        pop     bx                      ;restore buffer pointer
        dec     count                   ;subtract 1 from record counter
        cmp     count,0                 ;have we printed all records?
        jne     write_1                 ;no, go back and do next one
```

character

write ONE char

```
        mov     ah,3eh                  ;close file function
        mov     bx,output               ;this file handle
        int     21h                     ;call dos
        ret                             ;return to caller
write   endp
;=========================================================
;SAVE saves the current filename and its description to
;the MAC_REC buffer
;=========================================================
save    proc    near
        inc     count                   ;add one to record counter
        mov     di,place                ;DI=destination address
        mov     si,offset dta           ;SI=disk transfer buffer
        add     si,30d                  ;move up to ASCIIZ filename
        mov     cx,13d                  ;length of filename maximum
save_1: mov     al,[si]                 ;get one character of filename
        cmp     al,0                    ;end of filename reached?
        je      save_2                  ;yes, pad with spaces if short
        mov     [di],al                 ;save character in mac_rec
        inc     si                      ;bump both buffer
        inc     di                      ;pointers
        loop    save_1                  ;loop until filename copied
        jmp     save_3                  ;and continue
;If the filename is less than 12 bytes long, pad it with spaces
save_2: mov     al,20h                  ;get a space character
        mov     [di],al                 ;store in mac_rec
        inc     di                      ;bump buffer pointer
        loop    save_2                  ;until its padded with spaces
;Now put 5 spaces in the buffer to separate filename from description
save_3: mov     cx,5d                   ;pad record with 5 spaces
save_4: mov     al,20h                  ;get a space character
        mov     [di],al                 ;store in mac_rec
        inc     di                      ;bump buffer pointer
        loop    save_4                  ;until we have a 'tab' space
;Now go save the description in mac_rec as well
;Make a test to see if this macro file has a description
;attached to it
see:    mov     bx,offset databf        ;point to data
        add     bx,56d                  ;move up to descriptin byte
        mov     al,[bx]                 ;get the first byte there
        cmp     al,0                    ;is there a description?
        je      nodes                   ;no, use default then
        cmp     al,0dh                  ;is there a description?
        je      nodes                   ;no, use default then
;Next, send the description to the printer
out_3:  mov     si,offset databf        ;BX=the file's data
        add     si,56d                  ;move up to start of description
        mov     cx,40d                  ;maximum length possible for it
out_4:  mov     al,[si]                 ;get the character in DL
        cmp     al,0                    ;end of description reached?
        je      out_5                   ;yes, go process next macro file
        mov     [di],al                 ;save in mac_rec buffer
        inc     di                      ;bump buffer pointer
        inc     si                      ;and this one too
        loop    out_4                   ;get next character to process
out_5:  jmp     save_5                  ;go process next macro file
;This macro file does not have a description, print message
;'No description available' instead
nodes:  mov     si,offset nosuch        ;no description available
        mov     cx,40d                  ;length of message text
des_2:  mov     al,[si]                 ;get the character
        cmp     al,0                    ;end of message reached?
        je      out_5                   ;yes, keep going then
        mov     [di],al                 ;store in mac_rec buffer
        inc     di                      ;bump buffer pointer
        inc     si                      ;and this one too
        loop    des_2                   ;go back for another
        jmp     save_5                  ;do c/r and linefeed now
;Display the macro file's description
cont:   mov     si,offset databf        ;point to data
        add     si,56d                  ;move up to description byte
        mov     cx,40d                  ;show this many bytes maximum
show:   mov     al,[si]                 ;get one character
        cmp     al,0                    ;end of description reached?
        je      save_6                  ;yes, quit then
        mov     [si],al                 ;store the character in mac_rec
        inc     si                      ;bump buffer
        inc     di                      ;pointers
        loop    show                    ;get next one
        jmp     save_5                  ;go if descrip is 40 bytes long!
save_6: mov     al,20h                  ;pad description with spaces
save_7: mov     [di],al                 ;store the space in mac_rec
        loop    save_7                  ;until description padded
save_5: mov     di,place                ;get previous mac_rec position
        add     di,57d                  ;add length of one record to it
        mov     place,di                ;save new position for next time
        ret                             ;return to caller
save    endp
;======================================================================
;SORT sorts the records in MAC_REC into alpha order by filename
;======================================================================
sort    proc    near
        mov     si,offset mac_rec       ;records to be sorted
        mov     di,si                   ;into DI as well
        mov     dx,57d                  ;length of one record
        add     di,dx                   ;DI=set to second record
        mov     cx,count                ;number of records in CX
        dec     cx                      ;subtract one to begin with
sort_1: push    cx                      ;save record counter
        push    si                      ;save SI and DI
        push    di                      ;buffer pointers
sort_2: push    cx                      ;save record count
        mov     cx,dx                   ;prepare to compare two records
        push    si                      ;save SI
        push    di                      ;and DI
        rep cmpsb                       ;do the compare now
        pop     di                      ;restore DI
        pop     si                      ;and SI
        jbe     sort_3              ;continue if already in sort order
        mov     cx,dx               ;else swap the two records
        push    ax                  ;in the buffer
        push    bx                  ;save
        push    cx                  ;everything
        pushf                       ;including flags
        mov     bx,0                ;zero BX to start
sort_4: mov     al,[si+bx]          ;get source byte
        xchg    al,[di+bx]          ;put in second place
        xchg    al,[si+bx]          ;put in first place
        inc     bx                  ;bump buffer pointer
        loop    sort_4              ;loop until records swapped
        popf                        ;recover flags
        pop     cx                  ;and
        pop     bx                  ;everything
        pop     ax                  ;else
sort_3: add     si,dx               ;move up to next record
        add     di,dx               ;into DI
        pop     cx                  ;recover record count
        loop    sort_2              ;and back again
        pop     di                  ;and do
        pop     si                  ;all the
        pop     cx                  ;records in
        loop    sort_1              ;the buffer
        ret                         ;return to caller
sort    endp
codesg  ends                        ;end of code segment
        end     start               ;end of program
```

Chapter 3

DIRNAME

DIRNAME is used to rename subdirectories. This operation can normally be done only through the use of the DOS 'SHELL' program.

In the previous chapter, you learned how to manipulate directories and files, redirect data to a file or device, sort a file of data in memory, get input from the keyboard and a number of other useful functions.

Some of these same code fragments and techniques will be used in the program DIRNAME. In addition, new routines will be introduced to show you how to determine the version of DOS installed in a computer system, retrieve and examine command line parameters, search for files with specific attributes, and a simple method of attracting attention by sounding a bell on the PC's speaker.

Renaming Directories Easily

The utility presented in this section is called DIRNAME. It is a small program that will enable you to rename directories on a floppy or hard disk.

No matter which version of DOS is installed in your computer system, you cannot rename a directory easily. The one exception is if you are running DOS 5.0. This version of the operating system does provide a means of renaming directories, but you must be using the SHELL program. It seems silly to have to learn to load and use SHELL just to perform one simple task. This is where DIRNAME comes in handy.

Functions Used in DIRNAME.ASM

Int 21h, AH=09h	Display string
Int 21h, AH=1Ah	Set disk transfer address
Int 21h, AH=30h	Get DOS version
Int 21h, AH=4Ch	Terminate process with return code
Int 21h, AH=4Eh	Find first matching file
Int 21h, AH=4Fh	Find next matching file
Int 21h, AH=46h	Rename/move file

How to Use DIRNAME

To use DIRNAME, type the program's name at the DOS prompt followed by the name of the directory you want to rename and the new name to be given to the directory. For example, to rename a directory called OLDDIR on the current disk and give it the new name NEWDIR, type:

```
DIRNAME OLDDIR NEWDIR
```

The drive letter does not have to be specified if you are calling DIRNAME from the current default disk. If you want to rename a directory on another disk, then the drive must be specified.

After you press the ENTER key, DIRNAME will attempt to rename the specified directory. DIRNAME will not rename a directory if a directory with the new name already exists on the disk. In addition, you should not attempt to rename a directory if this is the current default directory—DOS may produce unpredictable and disastrous results if you do this. In all other cases, the directory will be renamed successfully.

Determining the DOS Version

Because DIRNAME uses a DOS function call available only under DOS 3.0 and later, DIRNAME's first task is to check the version of MS-DOS currently installed in the computer system. This is done with Function 30h of Int 21h, as shown below:

```
begin:  mov     ah,30h              ;get DOS version function
        mov     al,00h              ;to check
        int     21h                 ;call dos
        cmp     al,3                ;is it 3 or higher?
        jae     dos_ok              ;yes, continue
        mov     dx,offset baddos    ;no, DX=error message
```

```
b_exit: mov     ah,09h                  ;display string function
        int     21h                     ;call dos
exit:   mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
```

On return from this function call, the major version number is returned in the AL register and the minor version number is returned in the AH register. DIRNAME then checks to see if the major version number is 3 or greater.

If DIRNAME finds that the operating system fits its requirements, the program continues execution at the label DOS_OK.

If the computer system is running a DOS version less than 3.0, then an error message is displayed. The DX register holds the address of the error message we want to display, and Function 09h of Int 21h is called to output the string to the screen. Then the program is terminated by issuing an Int 21h, Function 4Ch call.

Command Line Parameters

A Word About DOS Version Function Calls

Programs that rely on certain features built into different versions of MS-DOS use Function 30h to determine the current version of DOS installed in the computer system. However, the DOS command SETVER was introduced in the new 5.0 release of the operating system. This command is used to fool the computer system into believing a different DOS version is installed.

If a user could have used the SETVER command, this could pose a major problem for a utility trying to determine the current operating system. To help circumvent this problem, Microsoft included, in DOS 5.0 and later, a new function call that can be used to determine the actual version of DOS in the system, regardless if the SETVER command has been used.

It is highly recommended that you use the new DOS Function 3306h if you need to be certain about the version installed in order for your program to work correctly.

Function 30h, Get Version Number

This function returns the version of DOS as set by the SETVER command.

To call:AL=00h	Original Equipment Manufacturer (OEM) Number or
AL=01h	DOS version flag
AH=30h	Function code
Returns:AL	Major version number
AH	Minor version number
BH	Version flag or OEM number. The version flag indicates if DOS is running in RAM or ROM. Only one bit in this byte is set to 1; all other bits are reserved and set to zero. If DOS is running in ROM, bit 08h is set to 1.
BL:CX	An optional 24-bit user serial number which is OEM dependent.

Function 3306h, Get MSDOS Version

This function returns the actual DOS version installed in the computer system, regardless if the SETVER command has been used.

To call:AH=33h	Function code
AL=06h	Sub-function code
Returns:BL	Major version number
BH	Minor version number
DL	The low 3 bits contain the revision number; all other bits are reserved and set to zero.
DH	Version flag. Only bits 08h and 10h indicate DOS is loaded in ROM and RAM, respectively.

When first executing DIRNAME, you must provide the original directory name and the new name you want to change it to. This can be done by taking advantage of command line parameters when the program is executed. In other words, the two directory names are specified at the DOS prompt as in: DIRNAME OLDDIR NEWDIR.

The method used to retrieve these parameters is called parsing the command line. To parse, or examine, the command line, a program looks inside its Program Segment Prefix. The PSP is created automatically by DOS each time a program is loaded into memory. This special area of memory allocated by the operating system is 256 bytes long. Each field in the PSP contains information needed by the currently running program. This information can be read at any time. It is, however, highly recommended that you not change any of the data stored in the PSP. (Appendix A shows the structure of the PSP).

Each time DOS loads a program into memory, whether it's a TSR or a regular executable, it copies the contents of the command line to the PSP for that program. The command line is always saved starting at offset 80h in the PSP, and is therefore available to the program for reading. A count of the number of characters contained on the command line, not including the terminating carriage return (0Dh) byte, is stored in the first byte of this area of the PSP, i.e., at offset 80h.

Since DIRNAME does have two directory names that can be specified on the command line, the routine starting at the label DOS_OK examines the contents of the command line to see if these names were indeed specified. To do this, the SI register is loaded with the offset address of the first byte of DIRNAME's PSP.

```
dos_ok: mov     si,80h              ;SI=command line
        cld                         ;string moves go forward
        lodsb                       ;get the byte there
        cmp     al,00h              ;any parameters?
        jne     get_1               ;yes, continue
        mov     dx,offset syntax    ;no, DX=error message
        jmp     short b_exit        ;display message and quit
```

In order to read the data stored in the PSP, the LODSB, load string byte, instruction is used. This instruction loads the byte stored at address DS:SI into the AL register. The SI register is then automatically incremented to point to the next address, depending on the setting of the Direction Flag (DF). In DIRNAME, we used the CLD instruction before entering this loop. Conversely, if we had wanted to decrement SI, we would have used the STD instruction first.

Since the first byte represents a count of the number of characters on the command line, we need to test the value now stored in the AL register. If the length is equal to zero, we know that the directory names were not specified at runtime. In this case, the program displays a syntax error message and the utility is terminated.

On the other hand, if the value in AL is one or greater, the program branches to the label GET_1, shown here:

```
get_1:  lodsb                          ;get next character
        cmp     al,20h                 ;is it a leading space?
        je      get_1                  ;skip all of them first!
        mov     di,offset orig_dir     ;DI=destination buffer
        stosb                          ;store the first byte of name
get_2:  lodsb                          ;get next character
        cmp     al,20h                 ;is it delimiter between names?
        je      get_3                  ;yes, continue then
        stosb                          ;no, save the byte in buffer
        jmp     short get_2            ;go back for another
get_3:  mov     al,0                   ;get a zero byte
        stosb                          ;create ASCIIZ string
        mov     di,offset new_dir      ;DI=destination buffer
```

The routine here reads the first parameter from the PSP and stores it in the ORIG_DIR buffer. First, a check is made to see if the character is a space (sometimes users inadvertently type more than one space between the program name and its command line parameters). If the character is indeed a space, the program loops back, ignoring all the leading space characters it finds. This is a very important step. If you don't skip the leading space characters, your first parameter will have them instead!

Once all the leading space characters have been skipped over and ignored, the DI register is loaded with the address of the ORIG_DIR buffer. The original directory name read from the command line will be stored in this buffer as an ASCIIZ string. As each byte is copied from the command line to the buffer, the character is tested to see if it is a space. A space is used to separate the two directory names entered on the command line, and this is used by DIRNAME to signal the end of the first directory name. As soon as the program encounters this delimiting character, the AL register is loaded with a zero byte, and this is stored as the last character in ORIG_DIR, thereby converting the directory name to an ASCIIZ string.

This entire process is then repeated for the second directory name, saving this string in the NEW_DIR buffer. The only difference this time is that the carriage return (0Dh) byte is used to signal the end of the directory name.

The Disk Transfer Address

After DIRNAME has saved the two directory names to the ORIG_DIR and NEW_DIR buffers, it must set its own Disk Transfer Address buffer before proceeding any further.

As was explained in Chapter 1, the Disk Transfer Address (DTA) is a block of memory allocated by DOS that acts as an I/O buffer for many file operations. In DIRNAME, the DTA is set to an internal buffer that is 128 bytes long, as shown here. From this point on, all directory searches will use this I/O buffer.

```
        mov     dx,offset dta       ;buffer for disk transfer area
        mov     ah,1ah              ;set DTA function
        int     21h                 ;call dos
```

DOS Function 1Ah, Set Disk Transfer Address, is used to tell DOS that a new DTA buffer is to be used as a holding area for all future I/O operations. The function code (1Ah) is loaded into AH and DS:DX holds the address of a buffer. In most cases, the DTA is changed only once during program execution.

Locating Directories and Files

After the DTA has been set to DIRNAME's own I/O buffer, the program branches to the label EXIST. Since it is not possible to rename a directory that does not exist, this code fragment attempts to locate the original directory on the disk. Remember, directories are simply files created with a special attribute that identifies this file as a directory.

```
exist:  mov     dx,offset orig_dir  ;DX=original dir name
        mov     cx,10h              ;directory attribute
        mov     ah,4eh              ;find first matching file
        int     21h                 ;call dos
is_it:  jc      no_find             ;error - no such directory
        jmp     short found         ;we found it
no_find:mov     dx,offset nodir     ;DX=error message
        jmp     b_exit              ;display it and exit to DOS
```

As was explained in the discussion about the MACLIST program presented in Chapter 1, DIRNAME relies on the Find First Matching File and Find Next Matching File functions. The only difference in DIRNAME is the attribute of the file we are looking for (specified in the CX register). In the case of DIRNAME, an attribute of 10h is used to retrieve only those files that represent directories on the disk.

If Function 4Eh was unable to find at least one matching file, then the Carry Flag will be set, indicating an error condition which is returned in AX, and the program branches to the label NO_FIND. This routine simply displays an error message and returns control to DOS.

On the other hand, if Function 4Eh was able to find a matching file, the program branches to the label FOUND. The DTA buffer will be filled in by DOS with information about the found file. To this end, you must always remember to use Function 1Ah of Int 21h to set the DTA to a buffer in memory reserved for this purpose before initiating a search call.

```
found:   mov      bx,offset dta          ;DX=directory entry
         add      bx,21d                 ;move up to attribute byte
         cmp      byte ptr [bx],10h      ;is it a directory?
         je       good_one               ;yes, continue
```

At this point, the BX register is loaded with the address of the DTA buffer. The Find File functions store information relating to this entry in the DTA, and we need to make absolutely sure that this matching file is indeed a directory entry. Unfortunately, when you ask the Find File functions to find directories, the service also returns all files that match the directory's name. In other words, the service will return normal files as well as directory files. Therefore, we must check the attribute byte as it appears in the DTA buffer. To do this, we add an offset of 21 bytes to BX and that byte is compared to the attribute of a directory entry. If this attribute is 10h, we know we have a directory entry and the program branches to the label GOOD_ONE. However, if the entry just found is not a directory, then the Find Next Matching File function is called to continue the search.

Changing Names

The last section of DIRNAME's source code actually does the work of renaming the directory. This code fragment starts at the label GOOD_ONE, shown here:

```
good_one: mov     dx,offset orig_dir     ;original name
          mov     di,offset new_dir      ;new name
          mov     ah,56h                 ;rename/move file function
          int     21h                    ;call dos
          jc      no_name                ;error - cannot rename!
          mov     dx,offset done         ;show message and quit
no_name:  mov     dx,offset noway        ;file not renamed
                  jmp      b_exit        ;show message and quit
```

To rename or move a file, DOS provides Function 56h of Int 21h. Starting with DOS version 3.0, this function can also be used to rename directories. Since the purpose of DIRNAME is to rename only directories, the program previously made sure that the correct version of the operating system was installed. Therefore, all we need to do is execute this service to rename our directory.

Function 56h requires three parameters on entry: the AH register holds the function number (56h); the DS:DX register pair holds the address of the original directory's name; and the ES:DI register pair holds the address of the new name to be given to the directory. Both strings addressed by DX and DI must be in ASCIIZ format.

After the function has been executed, the Carry Flag will be clear if the directory was renamed successfully. In this case, the program would display a message to this effect and terminate the program.

On the other hand, the Carry Flag will be set if the function was unable to rename the directory and AX will hold an error number. Again, DIRNAME would display an error message just before returning control to DOS.

Function 56h can return one of four possible error codes in the AX register. An error will be returned if the original directory (or file) or the path does not exist, or if the directory has a file attribute of read-only. In addition, if an attempt is made to move a file to another disk, an error will occur.

Making Little Noises

If you look closely at the error message labelled NODIR, you'll see a 07h byte inserted at the beginning of the string. When Function 09h of Int 21h encounters this byte, it sounds a bell on the PC's speaker. The 07h byte is not actually displayed on the video screen. You can use this technique to sound a bell at special times in your programs to attract attention to a particular message being displayed.

Another Way to Rename Files

As has been shown in this section, Function 56h of Int 21h can be used to rename a file or subdirectory. However, to rename a subdirectory using this service, the computer system must be running DOS version 3.0 or greater. This poses a problem for systems running earlier versions of DOS, but fortunately, DOS itself provides a solution.

There are, as you may already know, two methods DOS provides for manipulating files. The first method uses File Control Block structures while the second method uses file handles. One of the functions available since the first version of DOS, Function 17h of Int 21h, allows you to rename a file or subdirectory specified in an FCB structure.

To rename a file or directory, the address of an Extended File Control Block (EFCB) is passed to Function 17h. The table below shows the structure of an Extended FCB. It differs from a regular FCB in that the EFCB has a seven-byte header section appended to the beginning of the structure. One of these fields in the header can be used to set the attribute of the file you want to manipulate. In this case, the file's attribute would be 10h, that of a directory. In addition, Function 17h can be used to rename a Volume Label in exactly the same way a directory is renamed. You would, of course, use the file attribute of 08h when renaming Volume Label files.

Field Name	*Size*	*Description*
EFCB Flag	1 byte	Must have the value FFh to indicate this is an Extended FCB.
Reserved	5 bytes	Reserved by DOS.
Attribute	1 byte	The file's attribute.
Drive	1 byte	Drive code, where 0=default, 1=A, and so on.
Filename	8 bytes	The name of the file. If the filename is less than 8 characters long, it must be padded with space (20h) characters.
Extension	3 bytes	The filename's extension. If the extension is less than 3 characters long, it must be padded with space (20h) characters.
Block Number	1 word	Initially set to 0 by DOS when the file is opened. Each block consists of 128 records. The block number refers to the block that contains the current record being processed.
Record size	1 word	The size of a logical record. DOS uses the default size of 128.

File size	2 words	The size of the file in bytes.
File date	1 word	The date the file was created or last modified.
File time	1 word	The time the file was created or last modified.
Reserved	8 bytes	Reserved by DOS.
Current Record	1 byte	The current record number.
Random record	4 bytes	The random record number.

Function 17h requires only two parameters. The AH register holds the function code (17h) while DS:DX is set to the address of an Extended FCB structure. On return from this service, the AL register will be set to a value of zero if the function was successful. In the event an error occurred, then AL will be set to a value of FFh.

When renaming files through Function 17h, the address in DS:DX points to the Extended FCB structure. This structure must contain the original name of the directory you want to rename at offset 7 and the new name to be given to the directory at offset 24. It is important to note that the old directory name may contain an optional drive letter, but the new name must not.

In short, if you were to include the code necessary, you could make DIRNAME work with any version of DOS, simply by using Function 17h for older versions of the operating system.

Summary

DIRNAME, the utility presented in this section, has shown you how to:

- retrieve parameters specified on the command line
- read directory entries
- determine the DOS version installed in the computer system
- use file attributes to identify files versus directories
- rename a file or directory
- sound a bell on the PC's speaker

Projects

1. Rewrite DIRNAME so it uses Function 17h to rename the directory if the version of DOS installed in the computer system is less than 3.0. This would make the utility available for use on all DOS machines.
2. Add more error checking to DIRNAME to make the utility more user-friendly. As an example, at the label GOOD_ONE, Function 56h is called to rename the subdirectory. The only error checking done is with the JC NO_NAME statement. This routine at the label NO_NAME could be expanded to display separate error messages for each possible error condition (i.e., file not found, path not found, access denied and not the same device).

Program Listing

```
;DIRNAME.ASM
;<c> 1992 by Deborah L. Cooper
;
;This utility allows you to rename subdirectories on a floppy
;or hard disk. Syntax:  dirname [old dir name] [new dir name]
;Note, the directory name may include an optional drive letter
;
codesg  segment                         ;start of code segment
        assume cs:codesg,ds:codesg
        org     100h                    ;COM-style program

start:  jmp     begin                   ;skip over data area

baddos  db      0dh,0ah,'DIRNAME requires DOS 3.0 or greater'
        db      0dh,0ah,'to run!','$'
syntax  db      0dh,0ah,'DIRNAME [old dir] [nmew dir]','$'
orig_dir db     65 dup(0)               ;buffer for original name
new_dir db      65 dup(0)               ;buffer for new name
dta     db      128 dup(0)              ;disk transfer area
nodir   db      0dh,0ah,07h,'The specified directory does not'
        db      0dh,0ah,'exist - aborting program!','$'
done    db      0dh,0ah,'Directory was renamed','$'
noway   db      0dh,0ah,'Directory was NOT renamed','$'

;=============================================================
;Determine the DOS version installed. Function 56h requires
;DOS 3.0 or higher
;=============================================================
;
begin:  mov     ah,30h                  ;get DOS version function
        mov     al,00h                  ;to check
        int     21h                     ;call dos
        cmp     al,3                    ;is it 3 or higher?
        jae     dos_ok                  ;yes, continue
        mov     dx,offset baddos        ;no, DX=error message

b_exit: mov     ah,09h                  ;display string function
        int     21h                     ;call dos

exit:   mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
;
;============================================================
;Check the command line to see if a parameter was entered
;============================================================
;
dos_ok: mov     si,80h                  ;SI=command line
        cld                             ;string moves go forward
        lodsb                           ;get the byte there
        cmp     al,00h                  ;any parameters?
        jne     get_1                   ;yes, continue
        mov     dx,offset syntax        ;no, DX=error message
        jmp     short b_exit            ;display message and quit
;============================================================
;Retrieve the first directory name from the command line.
;Store this name in buffer ORIG_DIR
;============================================================
;
get_1:  lodsb                           ;get next character
        cmp     al,20h                  ;is it a leading space?
        je      get_1                   ;skipp all of them first!
        mov     di,offset orig_dir      ;DI=destination buffer
        stosb                           ;store the first byte of name!

get_2:  lodsb                           ;get next character
        cmp     al,20h                  ;is it delimiter between names?
        je      get_3                   ;yes, continue then
        stosb                           ;no, save the byte in buffer
        jmp     short get_2             ;go back for another

;============================================================
;Make this an ASCIIZ string
;============================================================
;
get_3:  mov     al,0                    ;get a zero byte
        stosb                           ;create ASCIIZ string
        mov     di,offset new_dir       ;DI=destination buffer

get_4:  lodsb                           ;get next character
        cmp     al,0dh                  ;C/R - end of command line?
        je      get_5                   ;yes, continue
        stosb                           ;no, save byte in buffer
        jmp     short get_4             ;go back for next one

get_5:  mov     al,0                    ;get a zero byte
```

```
        stosb                           ;create ASCIIZ string

        mov     dx,offset dta           ;buffer for disk transfer area
        mov     ah,1ah                  ;set DTA function
        int     21h                     ;call dos

;==============================================================
;Make sure this directory actually exists on the disk
;==============================================================
;
exist:  mov     dx,offset orig_dir      ;DX=original dir name
        mov     cx,10h                  ;directory attribute
        mov     ah,4eh                  ;find first matching file
        int     21h                     ;call dos

is_it:  jc      no_find                 ;error - no such directory
        jmp     short found             ;we found it

;==============================================================
;This specified directory does not exist
;==============================================================
;
no_find:mov     dx,offset nodir         ;DX=error message
        jmp     b_exit                  ;display it and exit to DOS

;==============================================================
;We found a matching file. Make sure its actually a
;directory as opposed to a data file or program file
;==============================================================
;
found:  mov     bx,offset dta           ;DX=directory entry
        add     bx,21d                  ;move up to attribute byte
        cmp     byte ptr [bx],10h       ;is it a directory?
        je      good_one                ;yes, continue

;==============================================================
;The file we just found is not a directory. Continue to
;look for other possible matching files
;==============================================================
;
        mov     dx,offset orig_dir      ;DX=directory to locate
        mov     cx,10h                  ;directory attribute
        mov     ah,4fh                  ;find next matching file
        int     21h                     ;call dos
        jmp     is_it                   ;and continue

;==============================================================
;Attempt to rename the directory now
;==============================================================
;
good_one: mov   dx,offset orig_dir      ;original name
        mov     di,offset new_dir       ;new name
        mov     ah,56h                  ;rename/move file function
        int     21h                     ;call dos
        jc      no_name                 ;error - cannot rename!
        mov     dx,offset done          ;show message and quit

no_name:mov     dx,offset noway         ;file not renamed
        jmp     b_exit                  ;show message and quit

codesg  ends                            ;end of code segment
        end     start                   ;end of program
```

Chapter 4

FILEFIND

Use FILEFIND to quickly find any file on a floppy or hard disk. This program is a prime example of recursive programming techniques.

The techniques involved in recursive programming are the main emphasis of this chapter. However, you will also learn how to change from one directory to another and to change the current disk to a new default drive. Along the way, a number of very interesting concepts are discussed. These include converting the case of characters, formatting filenames into fully qualified pathnames, and preserving memory for a file of records instead of creating a disk file.

Both MACLIST and DIRNAME used DOS functions to retrieve files from the directory, but FILEFIND goes one step further to demonstrate travelling up and down the directory tree as it searches for a file.

Finding Those Hidden Files

When hard disks for the IBM PC were first introduced years ago, everyone was amazed at how much data could be stored on them. But they soon realized the hard disk also presented a few problems. One of these problems is solved with FILEFIND.

FILEFIND will methodically search an entire hard disk in an attempt to find out where a particular file is stored. When it locates a file, it displays the full pathname of the directory where the file was found.

To programmers, FILEFIND is not only a useful utility to add to one's repertoire, but its also a prime example of the recursive programming technique.

Functions Used in FILEFIND.ASM

Int 10h, AX=02h	Clear screen
Int 10h, AH=0Eh	Display byte
Int 21h, AH=09h	Display string
Int 21h, AH=0Eh	Set drive
Int 21h, AH=19h	Get current drive
Int 21h, AH=1Ah	Set disk transfer address
Int 21h, AH=30h	Get DOS version
Int 21h, AH=3Bh	Set current directory
Int 21h, AH=47h	Get current directory
Int 21h, AH=4Ch	Terminate process with return code
Int 21h, AH=4Eh	Find first matching file
Int 21h, AH=4Fh	Find next matching file

How to Use FILEFIND

To use FILEFIND, type the program's name and a file specification at the DOS prompt. When you press the ENTER key, FILEFIND will methodically search the entire disk for the files that match your search specification. When a matching file is found, the file's name and the directory it was found in will be displayed on the screen.

The following are a few examples of executing FILEFIND.

FILEFIND A:*.*	Will locate all files stored on the disk in Drive A.
FILEFIND C:D*.DOC	Will locate all files stored in all directories on Drive C that start with "D" as the first letter of the filename and "DOC" as the filename extension.

Recursive Programming

FILEFIND is an excellent example of recursive programming. Recursive routines perform the same task an almost infinite number of times. The routine ends only when a special condition is met. In the case of FILEFIND, the routine would exit to DOS only when it has searched every directory, including nested subdirectories, and the files contained in these directories, for the target file(s).

How does FILEFIND do this? It first searches for subdirectories, starting from the root directory. If FILEFIND finds a subdirectory, that subdirectory then becomes the current working directory. If yet another subdirectory is found within this directory, FILEFIND makes that the new default directory. If this subdirectory does not contain another directory, then FILEFIND searches the current directory for the target file(s). When it has finished its search for the target file(s), it goes back to the previous subdirectory it found. This new directory is then searched to see if it also contains subdirectories. This routine continues until FILEFIND is unable to find any more subdirectories. This is what the term recursive means. The routine ends when only a certain condition is met; in this case, FILEFIND cannot find any more directories to search through for the target file(s).

FILEFIND uses the Find First Matching File and Find Next Matching File functions (4Eh and 4Fh, respectively) to retrieve the name of every file stored on the disk. A complete discussion of these two function calls can be found in Chapter 1. However, it's worth repeating here that you must remember to use Function 1Ah of Int 21h to set the DTA to a buffer in memory reserved as an I/O buffer before initiating a search call.

In Preparation for Searching

Like many other DOS utilities and TSR programs, FILEFIND expects to find a parameter (in this case a filename) on the command line when the program is executed. However, the filename may come in several formats. For example, the program could be started like these examples below:

```
FILEFIND WP.EXE
FILEFIND D:\WP.EXE
FILEFIND W*.*
```

or any other number of combinations.

As you may have already guessed, a very important amount of time and code will be dedicated to decoding the command line parameters before any disk searching is started by FILEFIND. You will see this, and many other routines, that are devoted simply to parsing command line parameters, saving certain system characteristics like the current video page and mode or the current directory. All of this needs to be done in preparation of a task that will temporarily change these conditions or settings when a program is executed.

Therefore, the first thing that must be done shortly after FILEFIND is executed is to parse the command line. This actually starts at the label BEGIN1.

```
begin1: mov     ah,19h              ;get current drive function
        int     21h                 ;call dos
        mov     drive,al            ;store for exit routine
```

Function 19h of Int 21h, Get Current Disk, returns a number representing the current default disk drive in the AL register. This number is set to zero for Drive A, 1 for Drive B, etc. Other DOS functions use drive codes beginning with 1 (i.e., 1=A, 2=B, and so on). Drive code zero is often used to specify the current default disk.

To continue, the drive code is saved to the variable DRIVE. Later on in the program, this drive code will be used to make certain that FILEFIND exits back to DOS at the original default drive and directory.

The next step FILEFIND does is to find out if a parameter was specified on the command line at execution time. DI is set to the address of the first byte of the command line which holds a count of the number of characters, if any, typed on the command line. To read this information, we use the code fragment:

```
        mov     di,82h              ;point to command line
        cmp     byte ptr [di-2],0   ;any parameters entered?
        je      nope                ;no, then exit to DOS
        jmp     parse               ;yes, continue
```

Here the compare instruction (CMP) is used to determine what the count value is. If the value at address 80h is 0, then no parameters were specified at runtime, and the program branches to the label NOPE, which simply displays an error message to the user and returns control to the operating system.

On the other hand, if the value at address 80h is non-zero, then the program continues execution at the label PARSE.

The code starting at the label PARSE determines if a disk drive is part of the parameter passed to the program. Here is the code that reads the command line parameter, byte by byte, to form a string in a way that FILEFIND can make use of:

```
parse:   mov     di,83h              ;is this byte a
         cmp     byte ptr [di],':'   ;colon?
         je      colon               ;yes, continue
default: mov     si,offset curdir    ;point to destination buffer
         mov     ah,47h              ;get directory name function
         mov     dl,0                ;for default drive
         int     21h                 ;call dos
         mov     si,82h              ;point SI to start of line
         jmp     sdirl               ;and continue
colon:   mov     si,82h              ;point to drive letter
         mov     al,[si]             ;move drive letter to AL
         and     al,5fh              ;convert to uppercase
         mov     bx,offset root      ;point to prefix
         mov     [bx],al             ;store drive letter there
         sub     al,'A'              ;convert drive to number
         mov     drive2,al           ;save for later
         mov     ah,0eh              ;select new drive function
         mov     dl,al               ;switch it to DL
         int     21h                 ;call dos
```

The above code's job is to produce a pathname based on whether or not a drive and directory was specified on the command line. This information is used to save the current drive and directory to a buffer called CURDIR. Then when FILEFIND has completed its work, the pathname and default drive stored in the variable DRIVE will be used to return to DOS at this particular location on the computer system. This is a good programming practice. All programs should exit back to DOS with everything the way it was prior to the program doing its own work.

Because FILEFIND's purpose is to locate a file or group of files anywhere on the disk, we know that both a drive and pathname, in addition to a filename, could be in the command line buffer. Therefore, if we compare the second byte (at offset 83h of the PSP) to the colon (:) character, we will know whether or not a drive letter was specified.

If a drive letter was not specified, then the program skips through to the label DEFAULT.

```
default: mov     si,offset curdir    ;point to destination buffer
         mov     ah,47h              ;get directory name function
         mov     dl,0                ;for default drive
         int     21h                 ;all dos
         mov     si,82h              ;point SI to start of line
         jmp     sdirl               ;and continue
```

DOS Function 47h is used to obtain and save the name of the current directory. Function 47h accepts two parameters: DS:SI holds the address of the I/O buffer where the directory name will be written after the call executes, and the DL register will hold the number of the drive to be used, where 0 specifies the default, 1=A, and so on. If the Carry Flag is clear, then Function 47h was successful and the buffer pointed to by DS:SI will contain the pathname. However, if the Carry Flag is set, then Function 47h will exit with an error code in the AX register. The only possible error condition is that an improper drive code was specified.

The pathname returned in the specified buffer by Function 47h is terminated with a zero byte. However, this pathname does not include the drive letter, colon and backslash characters. In addition, if the current directory is the root directory, then Function 47h inserts only a 00h byte as the pathname.

Since Function 47h doesn't return a fully qualified pathname complete with the drive letter, colon, backslash and pathname, FILEFIND must reconstruct the ASCIIZ string into a more useable form. The section of code starting at the label COLON does this work.

```
colon:  mov     si,82h              ;point to drive letter
        mov     al,[si]             ;move drive letter to AL
        and     al,5fh              ;convert to uppercase
        mov     bx,offset root      ;point to prefix
        mov     [bx],al             ;store drive letter there
        sub     al,'A'              ;convert drive to number
        mov     drive2,al           ;save for later
        mov     ah,0eh              ;select new drive function
        mov     dl,al               ;switch it do DL
        int     21h                 ;call dos
```

To format the pathname in the correct form, the drive letter is first retrieved from the command line and converted to uppercase. Next, the drive letter is changed to a drive number by subtracting the ASCII "A" character, which translates to 65d, from AL; this is saved in the variable DRIVE2 for later use. The last step in this section of the program is to log on to the designated drive where FILEFIND is to do its work.

Function 0Eh of Int 21h is used to tell DOS to set a new disk drive as the default drive. The function expects the drive code to be in DL before the call is issued.

Once the original drive and directory have been temporarily saved in memory and a new drive possibly selected, the next step is to get the filename, which could conceivably be a wildcard specification, and save it to the buffer FILEB. As you can see from the code fragment below, the file specification stored in FILEB is also converted to an ASCIIZ string in preparation for the Find File function calls. The filename is copied directly from the DOS command line using the instructions LODSB and STOSB.

```
        mov     si,84h                  ;DOS command line buffer
sdir1:  mov     di,offset fileb         ;destination
        mov     cx,12d                  ;maximum 12 bytes
get1:   lodsb                           ;get byte from SI
        cmp     al,0dh                  ;is it carriage return?
        je      get2                    ;yes, quit routine then
        stosb                           ;no, store the byte
        loop    get1                    ;until its copied out
get2:   mov     al,0                    ;get a terminating zero byte
        stosb                           ;and store it
```

Searching in All the Right Places

At this point in the program, a number of tasks have been done. The original disk drive and pathname have been saved so that when FILEFIND exits back to DOS, the system can be restored to its original state. In addition, the name of the file or group of files to be searched for has been copied from the DOS command line to the buffer FILEB. It's now time to begin the process of methodically looking through every directory on the designated disk for the target file(s).

To begin with, the Disk Transfer Address is set to FILEFIND's own buffer called DTA. This DTA buffer will hold the information returned by the Find File functions for each individual file it finds. The information includes the name of the file that was just found as well as other information needed by DOS so it can continue searching for other matching files.

Since FILEFIND makes extensive use of the Find First Matching File and Find Next Matching File functions, the next step is to set the DTA to its own buffer. All subsequent I/O done through FILEFIND will be directed to this new DTA.

Searching for Files

The actual file search routine starts at the label RFILE1. DX holds the address of the target file(s) to be looked for and the Find First Matching File function is called. At this point, only normal files, not directory files, are being searched for. If Function 4Eh does not find a file, the Carry Flag is set and the program branches to the label RFILE10.

```
        mov     dx,offset dta       ;point to temporary DTA buffer
        mov     ah,1ah              ;set new disk transfer area
        int     21h                 ;call dos
rfile1: mov     dx,offset fileb     ;go find first file in dir
        mov     ah,4eh              ;find first matching function
        mov     cx,0                ;normal file attributes only
        int     21h                 ;call dos
        jc      rfile10             ;go if no file found
```

On the other hand, if Function 4Eh did find a file that matches the name of the target file(s), the program continues executing at the label RFILE2, as shown below.

```
rfile2: mov     bx,offset dta       ;point to DTA buffer
        add     bx,30d              ;move up to filename
        mov     cx,12d              ;display 12-byte filename
rfile3: mov     al,[bx]             ;get one byte
        cmp     al,0                ;end of filename reached?
        je      rfile4              ;yes, so continue
        mov     ah,0eh              ;display byte function
        int     10h                 ;call bios
        inc     bx                  ;bump pointer
        loop    rfile3              ;until filename displayed
rfile4: mov     dx,offset msg5      ;ROOT directory
        mov     ah,09h              ;display string function
        int     21h                 ;call dos
```

This code fragment displays the filename just found by the Find File service, along with the name of the directory it was found in, which is the root directory. Notice that BX is loaded with the address of the current Disk Transfer Address and this pointer is incremented by 30 bytes. This makes BX point to the first byte of the filename within the DTA buffer. A simple loop using Function 0Eh of Int 10h, retrieves each byte of the filename and displays it on the screen. Each DOS filename can be up to 12 bytes in length. If the filename is less than 12 characters long, the loop procedure will abort itself when it encounters a zero byte in AL. The Find File functions terminate each filename with a zero byte, i.e., the filename is actually an ASCIIZ string.

The final step is to display the name of the directory where this file was found. Since FILEFIND processes all entries from the root directory to start with and then proceeds on from there to any and all possible subdirectories, it knows that this is the root directory. Therefore, MSG5 is displayed along with this filename.

The code starting at the label RFILE5 continues searching for the target file(s) using the Find Next Matching File function. If another file is found, the program loops back to the routine that displays the filename.

If no more matching files are found, however, the program branches to the label RFILE10.

Once FILEFIND has searched the root directory for all files matching the target filename, it's time to begin searching through all the subdirectories. As you may already know, DOS can have up to 32 levels of nested subdirectories. In other words, one single subdirectory branching from the root directory can itself contain 32 levels of additional subdirectories. It is this management scheme that FILEFIND must traverse in order to find its target files. A special buffer called DTABUFF is used to allow FILEFIND to process every possible directory it finds on the disk.

Located at the very end of the source code listing for FILEFIND, the DTABUFF is 39,000 bytes long. Although not all of this buffer will be used, its size definitely ensures that FILEFIND can store the DTAs for up to 32 levels of directories.

Before we go any further, let's recap what we know about DOS's Find File functions. The first time you use the Find First Matching File function, Function 1Ah of Int 21h must have been previously called to set the DTA buffer. When the Find File function returns, it places information relating to the found file into the DTA buffer. At this point, you can retrieve whatever data you want from this buffer, such as the name of the file. Then, if you want to find the next possible file, the Find Next Matching File function uses this same DTA buffer's data to locate subsequent files stored on the disk.

In simple terms, this means that the same DTA buffer is used to find any number of files, providing those files reside in only one specific subdirectory. Therefore, for each subdirectory that FILEFIND finds, it must save the current Disk Transfer Address's data to the DTABUFF buffer before it can change to the next level on the directory tree. This is the purpose of the DTABUFF buffer. As FILEFIND moves up and down the directory tree, it switches to the previous or next DTA's data stored in the DTABUFF. In short, it's the method used by the program to keep track of its location on the disk.

From now on, when FILEFIND finds an entry in the root directory that is actually a subdirectory, it will save the current Disk Transfer Address's data to the DTABUFF. Since each DTA buffer is 43 bytes in length, a pointer STADDR is used to mark the last location written to in DTABUFF. This is first done at the label RFILE10.

```
rfile10: mov     dx,offset dtabuff   ;point to buffer
         mov     ah,1ah              ;to be used as disk
         int     21h                 ;transfer area
         mov     staddr,dx           ;save as starting point
```

At this point in the program, we need to save the directory we are working with in the buffer PATHBF. This pathname must include the drive letter as well as the name of the directory in ASCIIZ format. To this end, the first three bytes of PATHBF must contain the characters "C:\". Note that the drive letter will differ depending on what disk drive FILEFIND is working with. The code fragment shown below does all of this preparation work.

```
         mov     bx,offset root      ;point to prefix
         mov     al,[bx]             ;get drive letter
         mov     bx,offset pathbf    ;point to buffer
         mov     [bx],al             ;store it
         inc     bx                  ;bump buffer pointer
         mov     al,':'              ;get a colon
         mov     [bx],al             ;store it
         inc     bx                  ;bump buffer pointer
         mov     al,'\'              ;get a backslash
         mov     [bx],al             ;store it
         inc     bx                  ;bump pointer
         mov     al,0                ;get a ASCIIZ terminator
         mov     [bx],al             ;store it
         jmp     first               ;skip over next routine
```

After the PATHBF buffer has been successfully formatted for later use, the program branches to the label FIRST, shown here:

```
first:   mov     dx,offset file      ;point to wildcard filespec
         mov     ah,4eh              ;find first matching file
         mov     cx,10h              ;that is a directory
         int     21h                 ;call dos
         jc      quit                ;exit if drive does not have file
isit:    mov     bx,staddr           ;get starting addr for this entry
         add     bx,21d              ;move up to file attribute code
         mov     al,[bx]             ;get file attribute in AL
         cmp     al,10h              ;is it actual subdirectory?
         jne     next                ;no, find next one then
         mov     bx,staddr           ;get starting addr for entry
         add     bx,30d              ;move up to filename
         mov     al,[bx]             ;get first byte of filename in AL
         cmp     al,'.'              ;is it a dot directory?
         je      next                ;yes, do not use this one!
         jmp     path                ;and continue
```

The object of this code fragment is to search the current directory for files that have their attributes set to 10h. An attribute of 10h tells DOS that this file is a subdirectory. Once a subdirectory has been found by the Find File function, BX is set to the address of the filename just found. If the first byte of the filename is equal to the dot (.) character, then this directory entry is skipped over. Dot-named directories are used by the operating system to indicate the parent and current directories and serve no purpose to FILEFIND, which is why they are ignored when the program finds them.

If the entry just found is a valid subdirectory, the program branches to the label PATH. This routine, shown below, uses Function 3Bh of Int 21h to switch to the subdirectory just found, making that the new and current default directory.

```
path:     mov     si,staddr             ;point to DTA area
          add     si,30d                ;move up to filename
          mov     dx,si                 ;transfer start of dirname to DX
          mov     ah,3bh                ;change directory function
          int     21h                   ;call dos
```

Function 3Bh requires only two parameters. The function code (3Bh) must be placed in AH, and the name of the directory you want to switch to is addressed with DX. Therefore, the address of the current DTA is retrieved into SI first, then an offset of 30 bytes is added which makes SI hold the address of the actual filename. This is transferred to DX as required by the service call. Once this function call is executed, this subdirectory becomes the new default directory.

The next step is to retrieve the actual directory name we have just moved to on the disk and save this pathname in the buffer PATHBF. This is done like this:

```
          mov     si,offset pathbf+3    ;point to destination buffer
          mov     ah,47h                ;get directory name function
          mov     dl,0                  ;for default drive
          int     21h                   ;call dos
```

The Get Current Directory service is called by placing the function code (47h) in AH, a drive code in DL, and SI pointing to a buffer to which the directory name will be written. Notice that SI actually points to the fourth byte of PATHBF. This is done because we have already prepared the first three bytes to contain the drive letter, followed by a colon and then a backslash character earlier in the program. The string in PATHBF will be displayed later on to show where the target file was found on the disk.

The next section of code that is executed simply uses both Find File functions to search the default directory for any files that match the target name. If a file is found, the filename and the pathname are displayed, as has already been discussed earlier. If no more files are found in this directory, the program branches to the label FILE10.

```
file10: mov     dx,staddr               ;get previous DTA address
        add     dx,128d                 ;move up to next position for it
        mov     staddr,dx               ;save for next time through
        mov     ah,1ah                  ;set disk transfer area function
        int     21h                     ;call dos
        jmp     first                   ;look for first dir in this one
```

The variable STADDR is updated to point to the next available address in the DTABUFF that will be used on subsequent searches for subdirectories. Again, the Set Disk Transfer Address service is called to make certain all disk I/O is written to the buffer at the offset address held in STADDR. Then the program branches back to the label FIRST.

As explained earlier, the whole process starting at the label FIRST is repeated continuously until all directories in the current directory have been searched for the target file.

When FILEFIND has searched the entire disk, the program branches to the label QUIT. Function 0Eh of Int 21h is called to return the system to the disk drive it was on when FILEFIND was first executed. The current directory is also restored using Function 3Bh and then the program is terminated back to DOS.

Summary

Although the job of finding files on a hard disk appears to be relatively simple from a programmer's viewpoint, FILEFIND shows that what appears to be easy is not always the case. This utility has shown:

- how recursive routines and procedures are developed
- how to find a file or group of files in a subdirectory
- file attributes and their uses
- how to convert characters to uppercase or lowercase
- how to change directories

Projects

1. Near the end of the program listing for FILEFIND.ASM is the statement:

   ```
   dtabuff dw * + 100h + 39000d  ;leave for normal stack
   ```

   This reserves room in memory for disk input and output functions. However, when the program is assembled, the final COM file contains 39,000 blank characters. This wastes disk space and makes the program unnecessarily large. Find a different method of reserving memory at runtime, which in turn will make the final COM file much smaller.

2. If FILEFIND locates a filename with a ZIP filename extension, it could also be written to search this compressed file for the target file(s) you are trying to locate. Add the routines to do this, since many people download ZIP files from local bulletin boards and commercial information services.

3. You could modify FILEFIND to create a new utility that would delete all the target files it finds. To be safe, it should prompt you for a Yes/No response before actually removing the file(s) from the disk.

4. Specify options to locate only those files created or last modified since a particular date. The command line parameter could be specified as: FILEFIND *.TXT /D=02/25/93.

For Further Study

PC Magazine published a utility for uncompressing ZIP archive files in the March 31, 1992, Volume 11, Number 6, issue.

Program Listing

```
;FILEFIND.ASM
;Utility to locate a file or group of files on a hard drive
;syntax: FILEFIND [drive][wildcard filename]
;<c> 1992 by Deborah L. Cooper
;
codesg  segment                         ;start of code segment
        assume cs:codesg                ;tell MASM about it
        org     100h                    ;make this a COM file
start:  jmp     begin                   ;skip over data area
```

```
root    db      'c:\',0                 ;root directory pathname
fileb   db      12 dup(0),0             ;for searching for specific files
file    db      '*.*',0                 ;ASCIIZ wildcard filespec
bkdir   db      '..',0                  ;so we can move back one sub-dir
staddr  dw      0                       ;temporary holding address
pathbf  db      64 dup(0)               ;buffer for complete pathname
dta     db      128 dup(0)              ;disk transfer area buffer
msg     db      0dh,0ah,'$'             ;carriage return and linefeed
msg2    db      0dh,0ah,'File Finder syntax is:'
        db      0dh,0ah,'FILEFIND [drive]:[wildcard filespec]'
        db      0dh,0ah,'$'
msg4    db      ' in Directory: ','$'
msg5    db      ' in ROOT Directory',0dh,0ah,'$'
dosmsg  db      0dh,0ah,'File Finder requires MSDOS Version 2.0 or'
        db      0dh,0ah,'greater to execute - aborting to DOS','$'
copywr  db      0dh,0ah,'File Finder Utility'
        db      0dh,0ah,'<c> 1988 by Debbie Cooper',0dh,0ah,'$'
curdir  db      64 dup(0)               ;original starting directory
drive   db      0                       ;current default drive
drive2  db      0                       ;drive to be used for file search
begin:  mov     ax,02h                  ;clear the screen
        int     10h                     ;call bios
        mov     dx,offset copywr        ;program title
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,30h                  ;get MSDOS version function
        int     21h                     ;call dos
        cmp     al,2                    ;is it 2.0 or greater?
        jae     begin1                  ;yes, continue
        mov     dx,offset dosmsg        ;no, display error
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
begin1: mov     ah,19h                  ;get current drive function
        int     21h                     ;call dos
        mov     drive,al                ;store for exit routine
        mov     di,82h                  ;point to command line
        cmp     byte ptr [di-2],0       ;any parameters entered?
        je      nope                    ;no, then exit to DOS
        jmp     parse                   ;yes, continue

nope:   mov     dx,offset msg2          ;filefind help message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
;=================================================================
;See if a drive was specified on the command line
;=================================================================
parse:  mov     di,83h                  ;is this byte a
        cmp     byte ptr [di],':'       ;colon?
        je      colon                   ;yes, continue
```

```
;==========================================================
;If a drive was not specified, then use the current default
;==========================================================
default:mov     si,offset curdir        ;point to destination buffer
        mov     ah,47h                  ;get directory name function
        mov     dl,0                    ;for default drive
        int     21h                     ;call dos
        mov     si,82h                  ;point SI to start of command line
        jmp     sdir1                   ;and continue
colon:  mov     si,82h                  ;point to drive letter
        mov     al,[si]                 ;move drive letter to AL
        and     al,5fh                  ;convert to uppercase
        mov     bx,offset root          ;point to prefix
        mov     [bx],al                 ;store drive letter there
        sub     al,'A'                  ;convert drive to number
        mov     drive2,al               ;save for later
        mov     ah,0eh                  ;select new drive function
        mov     dl,al                   ;switch it to DL
        int     21h                     ;call dos
;==========================================================
;Save the original directory name as well
;==========================================================
sdir:   mov     si,offset curdir        ;point to destination buffer
        mov     ah,47h                  ;get directory name function
        mov     dl,0                    ;for new default drive
        int     21h                     ;call dos
;==========================================================
;Now set up the FILE buffer with the names we are to search for
;==========================================================
        mov     si,84h                  ;DOS command line buffer
sdir1:  mov     di,offset fileb         ;destination
        mov     cx,12d                  ;maximum 12 bytes
get1:   lodsb                           ;get byte from SI
        cmp     al,0dh                  ;is it carriage return?
        je      get2                    ;yes, quit routine then
        stosb                           ;no, store the byte
        loop    get1                    ;until its copied out
get2:   mov     al,0                    ;get a terminating zero byte
        stosb                           ;and store it
;==========================================================
;This is the routine to search through the current directory we are
;now in to find all files that match those specified by the user
;==========================================================
        mov     dx,offset dta           ;point to temporary DTA buffer
        mov     ah,1ah                  ;set new disk transfer area
        int     21h                     ;call dos
rfile1: mov     dx,offset fileb         ;go find first file in directory
        mov     ah,4eh                  ;find first matching function
        mov     cx,0                    ;normal file attributes only
        int     21h                     ;call dos
        jc      rfile10                 ;go if no file found
rfile2: mov     bx,offset dta           ;point to DTA buffer
        add     bx,30d                  ;move up to filename
        mov     cx,12d                  ;display twelve-byte filename
```

```
rfile3: mov     al,[bx]               ;get one byte
        cmp     al,0                  ;end of filename reached|
        je      rfile4                ;yes, so continue
        mov     ah,0eh                ;display byte function
        int     10h                   ;call bios
        inc     bx                    ;bump pointer
        loop    rfile3                ;until filename displayed
rfile4: mov     dx,offset msg5        ;ROOT directory
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
rfile5: mov     dx,offset fileb       ;go find next matching file
                                          ;in directory
        mov     ah,4fh                ;find next matching function
        mov     cx,0                  ;normal file attributes only
        int     21h                   ;call dos
        jc      rfile10               ;go if no file was found
        jmp     rfile2                ;else display the filename
rfile10:mov     dx,offset dtabuff     ;point to buffer
        mov     ah,1ah                ;to be used as disk transfer
        int     21h                   ;area
        mov     staddr,dx             ;save as starting point
;=========================================================
;Switch to the root directory of this drive
;=========================================================
        mov     dx,offset root        ;point to root directory name
        mov     ah,3bh                ;change directory function
        int     21h                   ;call dos
;=========================================================
;Set up PATHBF so it has leading drive data
;=========================================================
        mov     bx,offset root        ;point to prefix
        mov     al,[bx]               ;get drive letter
        mov     bx,offset pathbf      ;point to buffer
        mov     [bx],al               ;store it
        inc     bx                    ;bump buffer pointer
        mov     al,':'                ;get a colon
        mov     [bx],al               ;store it
        inc     bx                    ;bump buffer pointer
        mov     al,'\'                ;get a backslash
        mov     [bx],al               ;store it
        inc     bx                    ;bump pointer
        mov     al,0                  ;get a ASCIIZ terminator
        mov     [bx],al               ;store it
        jmp     first                 ;skip over next routine
quit:   mov     ah,0eh                ;select drive function
        mov     dl,drive              ;to original drive
        int     21h                   ;call dos
        mov     dx,offset curdir      ;point to original dir name
        mov     ah,3bh                ;change directory function
        int     21h                   ;call dos
        mov     ah,4ch                ;terminate program function
        int     21h                   ;call dos
```

```
;===========================================================
;Search for first subdirectory in current directory
;===========================================================
first:  mov     dx,offset file          ;point to wildcard filespec
        mov     ah,4eh                  ;find first matching file
        mov     cx,10h                  ;that is a directory
        int     21h                     ;call dos
        jc      quit                    ;exit if drive does not have a file
isit:   mov     bx,staddr               ;get starting address for this entry
        add     bx,21d                  ;move up to file attribute code
        mov     al,[bx]                 ;get the file attribute in AL
        cmp     al,10h                  ;is it actual subdirectory?
        jne     next                    ;no find next one then
        mov     bx,staddr               ;get starting address for this entry
        add     bx,30d                  ;move up to filename
        mov     al,[bx]                 ;get first byte of filename in AL
        cmp     al,'.'                  ;is it a dot directory?
        je      next                    ;yes, do not use this one!
        jmp     path                    ;and continue
next:   mov     dx,offset file          ;point to wildcard filespec
        mov     ah,4fh                  ;search for next matching file
        mov     cx,10h                  ;that is a directory
        int     21h                     ;call dos
        jnc     isit                    ;go if we found a file that matched
        mov     dx,offset bkdir         ;move back one directory place
        mov     ah,3bh                  ;change directory function
        int     21h                     ;call dos
;===========================================================
;If we are already in the root directory, then we would get an
;error message except in this case, we have finished searching
;the entire drive so exit back to DOS
;===========================================================
        jc      quit                    ;exit to MSDOS now
        mov     bx,offset root          ;point to drive prefix
        mov     al,[bx]                 ;get drive code in AL
        mov     bx,offset pathbf        ;point to destination buffer
        mov     [bx],al                 ;store it
        inc     bx                      ;bump pointer
        mov     al,':'                  ;get a colon
        mov     [bx],al                 ;store it
        inc     bx                      ;bump pointer
        mov     al,'\'                  ;get a slash
        mov     [bx],al                 ;store it
        inc     bx                      ;bump pointer
        mov     al,0                    ;get a ASCIIZ terminator
        mov     [bx],al                 ;store it
        mov     si,offset pathbf+4      ;point to start of pathname
        mov     ah,47h                  ;get current pathname function
        mov     dl,0                    ;for default disk drive
        int     21h                     ;call dos
cont2:  mov     bx,staddr               ;get last DTA address used
        sub     bx,128d                 ;go back to previous DTA now
        mov     staddr,bx               ;save for next possible time through
        mov     dx,staddr               ;get start of DTA to be used
        mov     ah,1ah                ;set dta function
        int     21h                   ;call dos
        jmp     next                  ;go search next directory
path:   mov     si,staddr             ;point to DTA area
        add     si,30d                ;move up to filename
        mov     dx,si                 ;transfer start of dirname to DX
        mov     ah,3bh                ;change directory function
        int     21h                   ;call dos
;=======================================================
;Now set PATHBF equal to the new directory pathname
;=======================================================
        mov     si,offset pathbf+3    ;point to destination buffer
        mov     ah,47h                ;get directory name function
        mov     dl,0                  ;for default drive
        int     21h                   ;call dos
;=======================================================
;This is the routine to search through the current directory we are
;now in to find all files that match those specified by the user
;=======================================================
        mov     dx,offset dta         ;point to temporary DTA buffer
        mov     ah,1ah                ;set new disk transfer area
        int     21h                   ;call dos
file1:  mov     dx,offset fileb       ;go find first file in directory
        mov     ah,4eh                ;find first matching function
        mov     cx,0                  ;normal file attributes only
        int     21h                   ;call dos
        jc      file10                ;go if no file found
file2:  mov     dx,offset msg         ;show a c/r and linefeed
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        mov     bx,offset dta         ;point to DTA buffer
        add     bx,30d                ;move up to filename
        mov     cx,12d                ;display twelve-byte filename
file3:  mov     al,[bx]               ;get one byte
        cmp     al,0                  ;end of filename reached|
        je      file4                 ;yes, so continue
        mov     ah,0eh                ;display byte function
        int     10h                   ;call bios
        inc     bx                    ;bump pointer
        loop    file3                 ;until filename displayed
file4:  mov     dx,offset msg4        ;in directory...
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        mov     bx,offset pathbf      ;point to pathname
        mov     cx,64d                ;length of pathname (max)
file12: mov     al,[bx]               ;get one byte
        cmp     al,0                  ;end of pathname reached?
        je      file5                 ;yes, continue then
        mov     ah,0eh                ;no, display byte
        int     10h                   ;call bios
        inc     bx                    ;bump pointer
        loop    file12                ;until pathname displayed
file5:  mov     dx,offset fileb       ;go find next matching file
                                          ;in directory
        mov     ah,4fh                ;find next matching function
        mov     cx,0                  ;normal file attribute only
        int     21h                   ;call dos
        jc      file10                ;go if no file was found
        jmp     file2                 ;else display the filename
;=========================================================
;Now that we are in the subdirectory, we need to set the
;new DTA address to the next block (128 bytes)
;=========================================================
file10: mov     dx,staddr             ;get previous DTA address
        add     dx,128d               ;move up to next position for it
        mov     staddr,dx             ;save for next time through
        mov     ah,1ah                ;set disk transfer area function
        int     21h                   ;call dos
        jmp     first                 ;look for first directory in
                                          ;this one
showdir proc    near                  ;subroutine to display directory
                                          ;name
        mov     bx,offset pathbf      ;load BX with address of pathname
        mov     cx,64d                ;length of pathname (max)
showd1: mov     al,[bx]               ;get one byte
        cmp     al,0                  ;end of pathname reached?
        je      showd2                ;yes, so exit subroutine then
        mov     ah,0eh                ;display byte function
        int     10h                   ;call bios
        inc     bx                    ;bump buffer pointer
        loop    showd1                ;until entire pathname displayed
showd2: mov     al,0dh                ;get a carriage return
        int     10h                   ;display it
        mov     al,0ah                ;get a linefeed
        int     10h                   ;display it
        ret                           ;return to caller
showdir endp                          ;end of subroutine
dtabuff dw      $ + 100h + 39000d     ;leave for normal stack
codesg  ends                          ;end of code segment
        end     start                 ;end of program
```

Chapter 5

TRAPBOOT

TRAPBOOT prevents a user from accidentally rebooting the computer system by disabling the keystroke combination Ctrl+Alt+Del.

Up to this point, the previous three programs presented in this book have been developed as standard COM files. However, its time to learn how terminate and stay resident (TSR) programs are developed. In this and the remaining chapters, several TSR utilities will be discussed. In each case, a number of different techniques for developing TSRs will be demonstrated.

TRAPBOOT is the smallest and most easily implemented TSR presented in this book. Through this program, you will learn the basic building blocks of a memory resident program: how it is loaded into memory, how to monitor keyboard activity in order to activate TRAPBOOT, and preserving only the minimum amount of memory required to store the utility in RAM. In short, TRAPBOOT is an example of a small TSR program with all the sections necessary to make it memory resident.

Preventing Accidental System Reboots

Many new computer users (and sometimes even advanced gurus!) are frustrated when they accidentally tell the computer system to reboot itself. When the computer is asked to perform a reboot, all data stored in memory is lost forever—there is absolutely no way you can retrieve the information, unless, of course, it was previously saved to disk!

If this has ever happened to you, you'll know the value of having TRAPBOOT installed in your computer's memory. TRAPBOOT is a very simple example of a terminate and stay resident utility that resides in memory, monitoring all keys that are typed on the keyboard.

Whenever the keyboard routine detects that the Ctrl+Alt+Del key combination has been pressed, the computer performs a warm boot. However, when installed in memory, TRAPBOOT tells the computer to ignore this particular key combination, thereby preventing DOS from doing a warm reboot.

TRAPBOOT is the first memory resident program to be presented in this book. Although most TSR utilities are much more involved, you can get a good idea of how one is designed and implemented by studying the source code for TRAPBOOT.

Functions Used in TRAPBOOT.ASM

Int 21h, AH=09h	Display string
Int 21h, AH=25h	Set interrupt vector
Int 21h, AH=31h	Terminate and stay resident
Int 21h, AH=35h	Get interrupt vector
Int 21h, AH=49h	Release block of memory

How to Use TRAPBOOT

To run TRAPBOOT, type the program's name at the DOS prompt or put it in your AUTOEXEC.BAT file so it is installed in memory whenever you boot your computer system. TRAPBOOT needs less than 1K of RAM and should not interfere with other TSR utilities or application programs.

Once TRAPBOOT has been installed in memory, you will not be allowed to reboot the computer system by pressing the Ctrl+Alt+Del key combination. TRAPBOOT fools your computer into believing this key combination was not actually activated on the keyboard.

Making it Resident in Memory

Unlike regular DOS programs, TSR programs are actually divided into two sections. The first section of a TSR program is the code that stays resident in memory and it is located at the beginning of the program file. The second section is the code that loads the resident code into memory. This section, most often referred to as the transient code, is located at the end of the program file, and, precisely because of its location, its code and data areas are discarded when the program terminates back to DOS.

Although the transient portion of the TSR program is not made resident in memory, it is the first part of the TSR that actually gets executed. If you look at the first few lines of the source code for TRAPBOOT, the statement "BEGIN: JMP INIT" transfers control immediately to the transient portion of the program. This code, then, is responsible for putting the resident code into memory.

There are three DOS function calls that can be used to terminate a program. For regular DOS programs, Function 4Ch of Int 21h, Terminate Process with Return Code, is always used. After this call has been issued, control returns to the operating system. The memory used by the program is consequently released and can be used by the next program that is executed.

The other two DOS functions that terminate programs are usually only used in TSRs, Int 27h and Function 31h of Int 21h. Both functions reserve part of the computer's memory so that it will not be overlaid by the next program that is executed. However, it is preferable to use Function 31h of Int 21h because it allows a return code to be passed back to the parent process. This technique of passing return codes will be discussed shortly.

The code fragment below shows how TRAPBOOT is terminated:

```
mov     dx,(offset lastbyte + 15)/16
mov     ah,31h                  ;Terminate and stay resident
mov     al,0                    ;return code
int     21h                     ;call dos
```

Function 31h is the DOS routine that loads and saves TRAPBOOT's resident code in memory. This function expects DX to hold a count of the number of paragraphs of memory to reserve. As stated earlier, a return code can also be used, if desired, by the program to indicate special exit conditions. In the case of TRAPBOOT, the return code specified in AL was zero. If AL=0, then it is assumed the function was successful; any other value would indicate that an error had occurred.

Return codes would be useful if the program was loaded via a batch file. The batch file statement IF ERRORLEVEL could then evaluate the return code and proceed accordingly. In addition, the parent process could evaluate the return code by issuing a call to Function 4Dh of Int 21h, Get Return Code.

How do we know how much memory should be reserved for the TSR utility? This question is relatively easy to answer. Function 31h requests that the DX register be set to a value indicating how many paragraphs of memory should be reserved. A paragraph is the term used to describe a set of 16 bytes of consecutive memory.

The statement MOV DX,(OFFSET LASTBYTE + 15)/16 calculates the total number of paragraphs we need to reserve for TRAPBOOT. If you look closely at the program listing, toward the end of the program is the statement LASTBYTE EQU $. The variable LASTBYTE was inserted in the code to indicate that everything from the beginning of the program up to this specific location is to be placed into memory by Function 31h. Therefore, all the code that appears after the LASTBYTE line does not become resident in memory. It is discarded once the utility becomes resident.

Once we know how many bytes should be reserved in memory, the expression MOV DX,(OFFSET LASTBYTE + 15)/16 is calculated when the program is assembled. This means that DX will be equal to the number of bytes up to the label LASTBYTE, plus 15 more bytes. The "/16" means to divide this value by 16 (the length of one paragraph), which in turn converts the final value in DX to the number of paragraphs of memory needed by TRAPBOOT.

Let's Interrupt for a Moment

The IBM PC and its compatibles generate hardware interrupts whenever the system detects activity on the keyboard, disk drive or other component. For example, whenever a keystroke is detected by the keyboard controller chip, a hardware interrupt is generated. This hardware interrupt gets set to the service routine for Int 09h. This service routine reads the scan code from the keyboard. The scan code is then converted into a key code and an ASCII code, depending on the state of the shift and toggle keys. It is this interrupt vector that TRAPBOOT chains into so that it can be called into action when the Ctrl+Alt+Del key combination has been pressed on the keyboard.

The interrupt vector routine for the keyboard and other system resources is simply an address. This address tells the computer where the code to handle this particular interrupt is stored in memory. By changing the keyboard interrupt vector address, TRAPBOOT can redirect this interrupt to its own service handler to determine if the Ctrl+Alt+Del key sequence was pressed.

Chaining Interrupt Routines

The technique of adding new routines to an already existing interrupt routine is called chaining into an interrupt vector. This means that your program redirects a specific vector to point to your program's routine. Once your program's routine has been executed, it calls the original interrupt service routine (ISR) to perform as usual. It is permissable to have many routines chaining into a single interrupt vector. You are limited only by the amount of memory you have installed in your computer system, although some TSR programs, if not written correctly, may interact with each other and crash the computer system.

A few simple rules must be followed when attempting to chain interrupt vectors. It is important to remember that, when calling the original routine, this must be done by a **far** CALL. The original routine will in turn perform a **far** return with the IRET instruction. The IRET instruction pops an offset and segment address and a flag register from the stack. This is done because the new interrupt service routine that performs a far CALL pushes a segment and offset address onto the stack and these addresses and flag must be removed to keep the stack properly managed. If this is not done correctly, the computer system will eventually crash.

Now, to actually hook into the keyboard interrupt handler, TRAPBOOT uses two very important DOS functions. The first function, Get Interrupt Vector, Function 35h of Int 21h, is used to retrieve the address of the current interrupt service routine for the interrupt specified in the AL register. The segment and offset address of the service routine is returned in ES:BX. The address returned in ES:BX is usually saved into a double-word variable so that when the program ends, the original interrupt service routine's vector can be restored. In the case of TRAPBOOT, this information is saved so that if the keystroke is not Ctrl+Alt+Del, control can be passed back to the original keyboard interrupt routine chained into memory.

The second DOS function used is Function 25h of Int 21h, Set Interrupt Vector. It is used to place a new routine into the chain of interrupt vectors. In this case, the routine we want the keyboard vector to go to first is called TRAPKEYS. We would like this procedure to be executed every time a key is pressed on the keyboard. To put this routine in the chain, Function 25h requires two parameters. The AL register holds the interrupt number we want to modify and DS:DX holds the segment and offset address of the new routine that will service this interrupt. Since we want to chain into the keyboard vector, the AL register would be loaded with the value 09h.

Where did we get the value of 09h above? Every time a key is pressed on the keyboard, a hardware interrupt (Int 09h) is generated. This keyboard interrupt produces a one-byte value that represents one particular key on the keyboard. However, if an extended keyboard is attached to the computer system, a two-byte scan code is generated, where the first byte is always E0h.

TRAPBOOT's keyboard routine monitors every keystroke the system receives. But to accomplish this, the initialization portion of the program must put its own routine for processing keystrokes into the keyboard interrupt vector. The code fragment below shows how an interrupt vector is changed.

```
mov     ah,35h              ;get interrupt vector addresses
mov     al,9                ;for Int 09h (keyboard)
int     21h                 ;call dos
mov     old_kbd_int,bx      ;save the offset address
mov     old_kbd_int[2],es   ;and segment address
mov     ah,25h              ;set interrupt vector addresses
mov     al,9                ;for Int 09h (keyboard)
lea     dx,trapkeys         ;to our routine
int     21h                 ;call dos
```

TRAPBOOT works by taking over control of the keyboard and monitoring each keystroke the user presses. Therefore, if the user presses Ctrl+Alt+Del, TRAPBOOT fools the system into thinking that that key combination was not actually pressed.

Understanding the Keyboard

When the PC's microprocessor detects that a key was pressed or released on the keyboard, it sends a scan code to System Control Port A, a hardware port at address 60h. This scan code is a number (byte length) that uniquely represents a key on the keyboard. A program can read the scan code directly from Port 60h, the hardware interface to the keyboard. After the scan code is sent to this port, the keyboard interrupt 9h is executed. The service routine associated with Int 09h then processes this keystroke.

The actual keystroke is further broken down into two categories. A different scan code is generated depending on whether the key was pressed or released. When a key is pressed, the scan code is called a make code; when the key is released, it is called a break code. Make codes are always one-byte scan codes, whereas break codes are two-byte scan codes. The first byte is always set to E0h and the second byte is the scan code itself.

If a scan code is generated from one of the shift (Ctrl, Alt, or Shift) or toggle (CapsLock, ScrollLock, Insert, or NumLock) keys, a bit in the Shift Key Status byte of the BIOS Data Area is changed. The Shift Key Status bytes are located at addresses 0040:0017 and 0040:0018 within the BIOS Data Area.

If the scan code is generated from a key release, the system disregards it. Otherwise, the keyboard interrupt routine determines if the scan code is that of a special-purpose key such as a function key that doesn't have an ASCII character code assigned to it. If the scan code does indicate that it is a function key, an extended scan code is also produced. The first byte of the two-byte extended scan code is always set to zero and the other byte is the regular scan code assigned to that particular key on the keyboard. In addition, the interrupt routine determines if the key struck was a combination of keys, such as Ctrl+Alt+Del, PrtScrn or SysReq, to name a few. In this case, the keyboard handler executes another interrupt routine to process these keystroke combinations.

Reading the Keyboard

The code fragment shown here is the heart of TRAPBOOT's main task, that of determining the status of the keyboard.

```
trapkeys proc   near                    ;trap Ctrl+Alt+Del key combo
         sti                            ;turn interrupts on
         push   ax
         push   ds
         in     al,60h                  ;read keyboard scan code into AL
         cmp    al,053h                 ;is the DELete key down?
         jne    not_ours                ;no, just exit to regular Int
         mov    ax,40h                  ;yes, AX=BIOS data segment
         mov    ds,ax                   ;transfer it to DS to can use it
         mov    al,ds:[17h]             ;AL=keyboard status byte
         mov    ah,al                   ;save in AH for now
         and    al,12                   ;test for bits 2 & 3 (ctrl+alt)
         cmp    al,12                   ;is Ctrl & Alt keys down?
         jne    not_ours                ;no, just exit
         and    ah,11110111b            ;tell BIOS Alt key not down
         mov    ds:[17h],ah             ;store in BIOS data segment
```

```
not_ours: pop    ds
          pop    ax
          cli                          ;turn interrupts off
          jmp    old_int_9h            ;go to original int 09h routine
trapkeys endp
```

To determine the status of the keyboard, the DS register is set to the BIOS data segment, which is located at address 40h in memory. Next, the instruction IN AL,60h is used to retrieve the scan code of the key just pressed. If that key's scan code is 53h, corresponding to the DELete key, then TRAPBOOT must make a further test for both the Ctrl and Alt keys.

If the key pressed is not the DELete key, TRAPBOOT passes the scan code back to the original keyboard service routine, as if nothing had happened. Indeed, all TSR programs that intercept keystrokes in this manner must make a call to the original routine to enable other application programs and TSRs to process the keystroke.

On the other hand, if the scan code indicates the DELete key was pressed, then TRAPBOOT's next task is to determine the state of the Ctrl and Alt keys. This could easily have been done by using Function 02h of Int 16h, but the direct access method was used in TRAPBOOT because this byte needs to be modified, as will soon be explained.

The keyboard status byte is the very same byte returned by Function 02h of Int 16h. In fact, a few interrupt functions can also be used to read many of the variables stored in the BIOS Data Area. Depending on which bits are set, a program can easily determine which of the shift keys has been pressed, as shown in the table below.

Shift Key Status Returned by Function 02h of Int 16h

Right shift down	0000 0001
Left shift down	0000 0010
Ctrl down	0000 0100
Alt down	0000 1000
Scroll Lock on	0001 0000
Num Lock on	0010 0000
Caps Lock on	0100 0000
Insert on	1000 0000

The BIOS Data Area holds information that the BIOS accesses when communicating with programs. Although most of the information stored in this special area of memory is readily accessible through interrupt functions, it is sometimes desirable for a program to access these variables directly, as has been done in TRAPBOOT.

Because the status of the keyboard is maintained by the BIOS whenever it detects a change, a program can determine if a certain toggle or shift key has been pressed simply by reading the Shift Key Status byte. Referring to the table shown above, if bit 2 is set, we know the Ctrl key is down; if bit 3 is set, the Alt key is down. Therefore, if both Ctrl and Alt are depressed, the value in AL will equal 12.

The next step for TRAPBOOT is to fool the BIOS into thinking that the three keys—Ctrl, Alt, and DELete—were not actually pressed at the same time. By resetting bit 3 in AL to zero, and saving this modified keyboard status byte back in the BIOS Data Area, the system is fooled into believing that this particular key sequence was not pressed in the first place.

Good Programming Practices

Program Listing 1 is the first version of TRAPBOOT that was developed. This version of the program requires 576 bytes of memory when installed. Version 2 of the same program only requires 496 bytes of memory. In Version 2, we simply moved the INSTALLED MESSAGE to the part of the program that does not remain in memory. After all, why save that string of text when it's only going to be used once when the program is installed? In other words, be aware of what code and data is being made resident in memory—perhaps some of it should be moved to the transient code section.

Version 3 of TRAPBOOT is further optimized to take advantage of the memory used by the PC. This code only consumes 320 bytes of memory. It achieves this by deallocating TRAPBOOT's copy of the environment and by requesting only the necessary amount of memory to be reserved that will hold the resident program. This is why you should take a good look at how you write yours TSRs; they must take advantage of every possible means to save code and data space. The practice of deallocating the environment block is a good one to get into using; many TSR authors do not bother to code this feature in, and hence their programs take up unnecessary space in memory. If you were to calculate the number of bytes saved for every TSR you write, you would eventually save enough space to be able to have another TSR in memory! By optimizing and changing a few things in the original version of TRAPBOOT, we have saved 256 bytes of precious random access memory—memory that can be used for other TSR programs or application programs.

Freeing Memory

TRAPBOOT includes a short routine that should be included in all TSR programs you write yourself. This routine deallocates TRAPBOOT's environment block. The environment block is inherited by the utility from the disk operating system (DOS) when the program is loaded and occupies 256 bytes of memory.

If your program does not need to access the environment block, then there is really no point in preserving this data in memory. The only exception to this rule of thumb would be if you wanted to see the name of the TSR as determined by some memory mapping utilities or DOS's MEM command.

The advantage to discarding the environment block is that you free up 256 bytes of memory. This may not be so significant on a system with extended or expanded memory, but it can, depending on the number of other TSR utilities loaded into memory, reduce an older PC's available memory. Also, by freeing the environment block, standard DOS applications have more memory to work with.

Function 49h of Int 21h, Release Memory Block, is used to deallocate the amount of memory reserved for the environment block. The segment address of the block of memory that you want to free is loaded into ES, with the AX register set to the number of paragraphs to be deallocated.

```
mov     ax,cs:[002ch]
mov     es,ax
mov     ah,49h              ;free block of memory function
int     21h                 ;call dos
```

The DOS Environment

The disk operating system maintains both a master environment and a current environment that can be accessed or changed by any programs you write. In fact, many configuration programs provide a facility for a program to modify its environment. For example, an installation program could modify the PATH statement to add a new directory to the list already present.

The master environment is created and used by DOS. If your configuration program needs to modify the environment it would do this in the master environment.

The current environment is created when DOS loads a program into memory. When the program is terminated, the current environment is then discarded. Therefore, the environment's variables are not usually modified in the current environment.

DOS stores the segment address of the current environment at offset 2Ch in the program's Program Segment Prefix (PSP). The actual data in this segment begins at offset 0. In addition, the master environment (i.e., the environment for COMMAND.COM) is found in the PSP as well.

If the Release Memory Block function was successful, the Carry Flag will be clear, otherwise it will be set. If an error has occurred and DOS was unable to deallocate the memory block, it usually means that DOS's chain of memory control blocks has somehow been corrupted. Therefore, your program should advise the user accordingly, perhaps with a message that he reboot his computer system to correct the problem.

Note that when calculating the amount of memory to release back to the system, the AX register contains this value in paragraphs. A paragraph is 16 bytes long and is computed when the program is assembled, not when the program is executed. The environment also includes the area called the Program Segment Prefix (PSP).

Summary

By examining the code in TRAPBOOT.ASM, a number of techniques were presented for creating terminate and stay resident programs. These techniques include:

- monitoring the keyboard for specific keystroke combinations
- changing and modifying interrupt service routines
- shrinking memory required by the TSR to a minimum in order to preserve the most memory possible for other TSR utilities and application programs

Projects

1. Optimize TRAPBOOT's code. This would make the program faster and its final size smaller. For example, instead of code like this:

   ```
   mov       ah,35h
   mov       al,9
   int       21h
   ```

   you could optimize these lines (and shorten the program listing!) as:

   ```
   mov       ax,3509h
   int       21h
   ```

2. Modify TRAPBOOT to disable other keystroke combinations like Ctrl+C and Ctrl+Break.
3. Instead of simply ignoring the keystroke combination Ctrl+Alt+Del, pop up a window on the screen that would ask the user if he really did wish to perform a reboot. If the response is "No," then restore the original contents of the screen before returning control to the application currently running when TRAPBOOT was invoked. If the response was "Yes," then do not restore the contents of the underlying application and perform the reboot immediately.

For Further Study

If you wish to learn more about optimizing programs that you write yourself, consult Michael Abrash's excellent book, *Zen of Assembly Language: Volume 1, Knowledge*. It is the definitive guide to optimizing assembly language code.

PC Magazine also published a 3-part series on optimizing programs written in assembly language. These articles appeared in Volume 10, issues Number 20, 21, and 22.

Program Listing 1

```
;TRAPBOOT.ASM
;Version 1.0
;<c> 1992 by Deborah L. Cooper
;
;This utility prevents the user from rebooting the computer by pressing
;CTRL+ALT+DEL key combination by mistake.
;
codesg  segment                       ;start of code segment
        assume  cs:codesg             ;set up CS for access
        org     100h                  ;make it a COM program

begin:  jmp     init                  ;go make us resident first

old_int_9h label dword                ;old Int 09h vector addresses
old_kbd_int     dw 2 dup(?)           ;is here

trapboot        proc    near          ;trap Ctrl+Alt+Del key combination
        sti                           ;turn interrupts on
        push    ax
        push    ds
        in      al,60h                ;read keyboard scan code into AL
        cmp     al,053h               ;is the DELete key down?
        jne     not_ours              ;no, just exit to regular Int 09h
then
        mov     ax,40h                ;yes, AX=BIOS data segment
        mov     ds,ax                 ;transfer it to DS to can use it
        mov     al,ds:[17h]           ;AL=keyboard status byte
        mov     ah,al                 ;save in AH for now
        and     al,12                 ;test for bits 2 & 3 (Ctrl+Alt)
        cmp     al,12                 ;is Ctrl & Alt keys down?
        jne     not_ours              ;no, just exit to regular Int 09h
then
        and     ah,11110111b          ;tell BIOS Alt key was not down
        mov     ds:[17h],ah           ;store it back in BIOS data segment
not_ours:
        pop     ds
        pop     ax
        cli                           ;turn interrupts off
        jmp     old_int_9h            ;go to original Int 09h routine
trapboot        endp                  ;end of our routine
;
lastbyte        equ $                 ;marker for end of resident code

msg     db      'TRAPBOOT <c> January, 1992 by Deborah L. Cooper'
        db      0dh,0ah,'has been installed','$'
;INIT is executed only once when the user initially installs TRAPBOOT
init    proc    near
        mov     ah,35h                ;get interrupt vector addresses
        mov     al,9                  ;for Int 09h (keyboard)
        int     21h                   ;call dos
        mov     old_kbd_int,bx        ;save the offset address
        mov     old_kbd_int[2],es     ;and segment address
        mov     ah,25h                ;set interrupt vector addresses
        mov     al,9                  ;for Int 09h (keyboard)
        lea     dx,trapboot           ;to our routine
        int     21h                   ;call dos
        mov     dx,offset msg         ;installed message
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
;Now make TRAPBOOT resident in memory. We must leave room in
;memory for the buffer, the stack and our program's code
        lea     dx,lastbyte           ;end of resident code
        int     27h                   ;terminate but stay resident
init    endp
codesg  ends                          ;end of code segment
        end     begin                 ;end of program
```

Program Listing 2

```
;===========================================================
;TRAPBOOT.ASM
;<c> 1992 by Deborah L. Cooper
;===========================================================
;
;This utility prevents the BIOS from rebooting the computer
;when the user presses the Ctrl+Alt+Del key combination @program =
;
codesg  segment                       ;start of code segment
        assume  cs:codesg             ;set up CS for access
        org     100h                  ;make it a COM program

begin:  jmp     init                  ;go make us resident first

old_int_9h label dword                ;old Int 09h vector addresses
old_kbd_int     dw 2 dup(?)           ;is here
;===========================================================
;TRAPKEYS is the actual routine that stays in memory. It
;determines if the Ctrl+Alt+Del key combination was pressed
;and if so, tells the BIOS to ignore it
;===========================================================
trapkeys proc    near                 ;trap Ctrl+Alt+Del key
combination
        sti                           ;turn interrupts on
        push    ax
        push    ds
        in      al,60h                ;read keyboard scan code into AL
        cmp     al,053h               ;is the DELete key down?
        jne     not_ours              ;no, just exit to regular Int 09h
then
        mov     ax,40h                ;yes, AX=BIOS data segment
        mov     ds,ax                 ;transfer it to DS to can use it
        mov     al,ds:[17h]           ;AL=keyboard status byte
        mov     ah,al                 ;save in AH for now
        and     al,12                 ;test for bits 2 & 3 (Ctrl+Alt)
        cmp     al,12                   ;is Ctrl & Alt keys down?
        jne     not_ours                ;no, just exit to regular Int 09h
then
        and     ah,11110111b            ;tell BIOS Alt key was not down
        mov     ds:[17h],ah             ;store it back in BIOS data segment
not_ours:
        pop     ds
        pop     ax
        cli                             ;turn interrupts off
        jmp     old_int_9h              ;go to original Int 09h routine
trapkeys endp                           ;end of our routine
;
lastbyte equ $-trapkeys+100h
;===========================================================
;The INIT routine is responsible for putting our interrupt
;service routine TRAPKEYS into memory
;===========================================================
author  db      'TRAPBOOT <c> January, 1992 by Deborah L. Cooper'
        db      0dh,0ah,'has been installed','$'

init    proc    near
        mov     ah,35h                  ;get interrupt vector addresses
        mov     al,9                    ;for Int 09h (keyboard)
        int     21h                     ;call dos
        mov     old_kbd_int,bx          ;save the offset address
        mov     old_kbd_int[2],es       ;and segment address
        mov     ah,25h                  ;set interrupt vector addresses
        mov     al,9                    ;for Int 09h (keyboard)
        lea     dx,trapkeys             ;to our routine
        int     21h                     ;call dos
        mov     dx,offset author        ;installed message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
;===========================================================
;The following routine deallocates or removes TRAPBOOT's
;copy of the environment block from memory
;===========================================================
        mov     ax,cs:[002ch]
        mov     es,ax
        mov     ah,49h                  ;free block of memory function
        int     21h                     ;call dos
;===========================================================
;The following calculates the number of paragraphs of
;memory that is required by TRAPBOOT
;===========================================================
        mov     dx,(offset lastbyte + 15)/16
;===========================================================
;The following code terminate our TRAPBOOT program,
;leaving the TRAPKEYS routine in memory
;===========================================================
        mov     ah,31h                  ;terminate and stay resident
function
        mov     al,0                    ;return code
        int     21h                     ;call dos
```

```
init    endp
codesg  ends                            ;end of code segment
        end     begin                   ;end of program
```

Program Listing 3

```
;TRAPBOOT.ASM
;<c> 1992, Deborah L. Cooper
;
;This utility prevents the user from rebooting the computer by pressing
;Ctrl+Alt+Del key combination by mistake.
;
codesg  segment                         ;start of code segment
        assume  cs:codesg               ;set up CS for access
        org     100h                    ;make it a COM program

begin:  jmp     init                    ;go make us resident first

msg     db      'TRAPBOOT <c> 1992 by Deborah L. Cooper'
        db      0dh,0ah,'has been installed','$'

old_int_9h label dword                  ;old Int 09h vector addresses
old_kbd_int     dw 2 dup(?)             ;is here

trapboot        proc    near            ;trap Ctrl+Alt+Del key combination
        sti                             ;turn interrupts on
        push    ax
        push    ds
        in      al,60h                  ;read keyboard scan code into AL
        cmp     al,053h                 ;is the DELete key down?
        jne     not_ours                ;no, just exit to regular Int 09h
then
        mov     ax,40h                  ;yes, AX=BIOS data segment
        mov     ds,ax                   ;transfer it to DS to can use it
        mov     al,ds:[17h]             ;AL=keyboard status byte
        mov     ah,al                   ;save in AH for now
        and     al,12                   ;test for bits 2 & 3 (Ctrl+Alt)
        cmp     al,12                   ;is Ctrl & Alt keys down?
        jne     not_ours                ;no, just exit to regular Int 09h
then
        and     ah,11110111b            ;tell BIOS Alt key was not down
        mov     ds:[17h],ah             ;store it back in BIOS data segment
not_ours:
        pop     ds
        pop     ax
        cli                             ;turn interrupts off
        jmp     old_int_9h              ;go to original Int 09h routine
trapboot        endp                    ;end of our routine
;
lastbyte        equ $                   ;marker for end of resident code
```

```
;INIT is executed only once when the user initially installs TRAPBOOT
init    proc    near
        mov     ah,35h                  ;get interrupt vector addresses
        mov     al,9                    ;for Int 09h (keyboard)
        int     21h                     ;call dos
        mov     old_kbd_int,bx          ;save the offset address
        mov     old_kbd_int[2],es       ;and segment address
        mov     ah,25h                  ;set interrupt vector addresses
        mov     al,9                    ;for Int 09h (keyboard)
        lea     dx,trapboot             ;to our routine
        int     21h                     ;call dos
        mov     dx,offset msg           ;installed message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
;Now make TRAPBOOT resident in memory. We must leave room in
;memory for the buffer, the stack and our program's code
        mov     dx,(offset lastbyte + 15)/16
        mov     ah,31h                  ;terminate and stay resident
        mov     al,0                    ;return code
        Int     21h                     ;call dos
init    endp
codesg  ends                            ;end of code segment
        end     begin                   ;end of program
```

Chapter 6

TRAPDEL

TRAPDEL is a utility that intercepts DOS's file delete function and places the to-be-deleted file in a special GARBAGE directory. It also demonstrates how DOS functions can be manipulated.

In the previous chapter, the TSR utility TRAPBOOT was developed to show you how TSRs are written. The basic format of a TSR is the same for each memory-resident program you may develop in the future. However, as you'll learn in this chapter, a TSR can become more complicated, requiring even more attention to making itself resident in memory while not disturbing the ordinary work of other foreground applications. The trick to writing well-behaved TSR programs is in taking it one step at a time, thoroughly debugging each section of the program as you progress in its development.

As mentioned earlier, TRAPBOOT is a very basic TSR program—that is, it's only a "shell" of a TSR. The program discussed in this chapter, TRAPDEL, is much more advanced from an assembly language point of view. In this discussion, you will learn how to intercept DOS service calls as opposed to monitoring keyboard activity. In addition, you'll also learn how to remove a TSR program from memory, and understand the difference between File Control Block and File Handle file access functions.

To Delete or Not

There are many times when you accidentally delete a file from the disk that really should not have been removed. TRAPDEL is a utility that prevents this from happening. It works by inserting its own routine into the interrupt vector table.

When a file is deleted, either at the DOS command prompt or through an application program, TRAPDEL steps in to the rescue and saves the file to another directory. All deleted files are moved to the subdirectory GARBAGE on the hard disk, although to DOS and application programs, it appears as though the file was indeed deleted.

By modifying software interrupt 21h, which processes DOS services, TRAPDEL is able to monitor all DOS function requests. It pops into action whenever it detects a request to delete a file from the disk.

When TRAPDEL detects that a request was made to delete a file, it calls its own procedure which moves the file to the GARBAGE directory. Then TRAPDEL makes a far call to the address of the original routine. This in effect allows TRAPDEL to manipulate the file before DOS actually tries to delete it from the disk. And, because TRAPDEL intercepts DOS interrupt 21h calls, its routine is completely transparent to the system.

Functions Used by TRAPDEL.ASM

Int 10h, AH=0Eh	Display byte
Int 21h, AH=09h	Display string
Int 21h, AH=1Ah	Set disk transfer address
Int 21h, AH=19h	Get default disk
Int 21h, AH=25h	Set interrupt vector
Int 21h, AX=3100h	Terminate and stay resident
Int 21h, AH=35h	Get interrupt vector
Int 21h, AH=39h	Create directory
Int 21h, AH=49h	Release block of memory
Int 21h, AX=4C00h	Terminate process with return code
Int 21h, AH=4Eh	Find first matching file
Int 21h, AH=4Fh	Find next matching file
Int 21h, AH=56h	Move/rename file

How to Use TRAPDEL

To run TRAPDEL, you simply type its name at the DOS prompt or put it in your AUTOEXEC.BAT file so it is installed in memory whenever you boot your computer system. TRAPDEL needs 1K of RAM and should not interfere with other TSR programs.

Once TRAPDEL has been installed in memory, you can issue the DOS "DEL" command to erase a file from the disk. DOS will report the file was deleted as usual. However, TRAPDEL has actually moved the file to a subdirectory named GARBAGE on your hard disk before DOS is allowed to "delete" the file from the disk.

In the event that an application program needs to delete a file, TRAPDEL will also intercept this command and move the file to the GARBAGE directory. In other words, whenever DOS functions are used to delete a file, TRAPDEL will be activated and DOS or the application program will be told the file was actually deleted even though it was just moved to a different subdirectory.

Making it Resident in Memory

When the operating system wants to delete a file from the disk, one of two functions may be used. The first is Function 13h of Int 21h, Delete File. This DOS service requires that DS:DX holds the segment and offset address of a File Control Block, which contains the name of the file to be deleted. The second DOS service used to delete files is Function 41h of Int 21h. However, this function requires that DS:DX holds the address of ASCIIZ pathname.

Although it is preferable in your own programs to use Function 41h to remove files from a disk, only one file can be deleted at any one time. This DOS service does not support the "*" and "?" wild card characters in the file specification. Function 13h, on the other hand, allows you to use the wild card characters to delete more than one file at a time.

Since we now know that DOS can use either of these services to delete a file, and both services are called via an Int 21h, TRAPDEL must intercept this interrupt. This is done in the initialization routine, starting at the label INIT, as shown here.

```
init    proc    near
        assume cs:codesg,ds:codesg
        cld                          ;clear DF first
        not word ptr start           ;destroy our first word's data
        xor     bx,bx                ;search from first segment
        mov     ax,cs                ;compare to this code segment
next:   inc     bx                   ;do this segment now
        cmp     ax,bx                ;until reaching our own
        mov     es,bx                ;code segment
        je      notyet               ;not installed yet
        mov     si,offset start      ;setup to compare the
```

```
         mov     di,si                  ;strings at DI and SI
         mov     cx,16                  ;compare this many bytes
         rep cmpsb                      ;do it now
         or      cx,cx                  ;did the strings match?
         jnz     next                   ;no, try next segment then
         mov     other_seg,es           ;yes, save TRAPDEL's segment
         jmp     not_in                 ;go check for commands
notyet:  mov     flag,1                 ;set flag to install later
```

The code fragment shown above determines if TRAPDEL has been previously installed in memory. This prevents you from installing multiple copies of TRAPDEL in your system and therefore saves precious memory space. All TSR programs should include this feature automatically.

To start the search, the Direction Flag (DF) is cleared to make sure that string comparisons go from left to right. In addition, the first two bytes of TRAPDEL's code (the one that is actually running now, not the "possible" TSR version) are modified to prevent a false match.

Since we want to start searching from the first possible block of memory, the BX register is set to zero. The AX register is then set to the address of TRAPDEL's code segment with the statement MOV AX,CS. The next set of statements beginning at the label NEXT compares the two strings, addressed by AX and BX, using the REP CMPSB instruction. If the two strings do not match, the program loops back to the label NEXT to repeat the process all over again. If the two strings do match, the address of the segment that contains TRAPDEL's program is saved in the variable OTHER_SEG. This segment address will be used later on in the initialization routine to remove this copy of TRAPDEL from memory if the "/U" switch is found on the command line. If this switch is not found, and if the variable FLAG is equal to a value of 1, then TRAPDEL does not try to install itself twice. This is a good preventative measure that your own TSRs should incorporate.

You may already have noticed that the program branches to the label NOT_IN regardless of whether copy of TRAPDEL resides in memory or not. This is done so that the command line parameters can be checked. If TRAPDEL is already installed in memory and the "/U" parameter is found on the command line, then we can go ahead and remove the program from memory since we have previously saved its segment address in the variable OTHER_SEG. On the other hand, if TRAPDEL is not installed yet, then TRAPDEL will be installed right away. By doing it this way, its easy to check for additional parameters on the command line. As an example, you may want to specify only certain files that should be actually deleted and not moved to the GARBAGE directory. Perhaps such files would have the filename extension BAK. This specification could be done just before TRAPDEL is made resident in memory.

Processing Command Line Switches

Since TRAPDEL has the feature of removing an already installed copy of itself from memory, you must have some way of telling TRAPDEL to do this. This can be done by specifying a command line switch, i.e., TRAPDEL /U where the "/U" characters are referred to as a switch or parameter.

Chapter 2 discusses how you can use the LODSB, load string byte, instruction in a loop routine to retrieve a filename from the DOS command line. This same general technique is used in TRAPDEL to determine if the "/U" parameter was specified but the routine is slightly modified to process this possible parameter, as you can see from the code fragment below.

```
not_in: mov     si,80h              ;address of command line
        lodsb                       ;get length of it in AL
        cmp     al,0                ;are there any switches?
        jz      make_r              ;no, go install then
com_2:  lodsb                       ;get next byte from line
        cmp     al,0dh              ;is it a carriage return?
        je      make_r              ;yes, exit this loop then
        cmp     al,20h              ;is it a space?
        je      com_2               ;yes, skip all leading spaces
        cmp     al,'/'              ;is it a slashbar?
        jne     make_r              ;no, go install now
        lodsb                       ;yes, get next byte
        and     al,5fh              ;convert to uppercase
        cmp     al,'U'              ;want to un-install now?
        jne     make_r              ;no, go install it
        jmp     uninstall           ;yes, go remove it
```

This routine retrieves the length of the command line from offset 80h within the PSP. If the value in AL is 0, then we know that no parameters were entered on the command line and the program continues on to the label MAKE_R. If the value in AL is not 0, then the program branches to the label COM_2.

Since we know that a parameter was typed on the command line, we need to retrieve it. To do this, we again use the LODSB instruction to retrieve each character from the command line one at a time. If this character is a carriage return (0Dh), then we know that the end of the command line was reached and control continues on to the label MAKE_R. In addition, all leading space (20h) bytes are skipped over (the user just typed too many spaces between the program's name and its parameters). If no parameters were found, then the program branches to the label MAKE_R, which installs TRAPDEL in memory.

On the other hand, if the backslash ("\") character is found, the next LODSB instruction retrieves the character following the backslash. Then we convert this character, now stored in the AL register, to uppercase. This is done to save us from having to make two comparisons, one for a lowercase "u" and one for an uppercase "U." If this character is a "U," TRAPDEL branches to the label UNINSTALL to attempt to remove itself from memory. If the character is not a "U," then we assume TRAPDEL should be installed for the first time and the program branches to the label MAKE_R.

At the label MAKE_R, the contents of the variable FLAG are compared to a constant value of one. If FLAG is equal to one, then we know that a copy of TRAPDEL already resides in memory. To this end, an appropriate error message is displayed using Function 09h of Int 21h, and the program is terminated. This simple check prevents us from installing a second copy of TRAPDEL in memory.

If the value in FLAG is equal to zero, the program continues executing at the label PUT_R.

```
make_r: cmp     flag,1                  ;install it now?
        je      put_r                   ;yes, go do it
        mov     dx,offset msg3          ;no, already here message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
put_r:  mov     ah,35h                  ;get interrupt vector function
        mov     al,21h                  ;for DOS functions
        int     21h                     ;call dos
        mov word ptr old21h,bx          ;save offset address
        mov word ptr old21h[2],es       ;save segment address
        mov     ah,25h                  ;set interrupt vector function
        mov     al,21h                  ;for DOS functions
        mov     dx,offset new21         ;to our NEW21 routine
        int     21h                     ;call dos
        mov     dx,offset author        ;copyright notice
        mov     ah,09h                  ;display sting function
        int     21h                     ;call dos
```

Monitoring Int 21h Functions

TRAPDEL's own procedure for processing Int 21h function calls must be inserted into the table of interrupt vectors stored in memory. The address of the old routine that processes Int 21h service calls is first retrieved and then saved in the variable OLD21H. To accomplish this task, Function 35h of Int 21h, Get Interrupt Vector, is used to retrieve the interrupt vector's segment and offset address. This address is returned in ES:BX and we save the address in the variable OLD21H for later use by TRAPDEL.

Function 25h of Int 21h is then used to set the Int 21h service to TRAPDEL's own procedure. The DS:DX register pair is loaded with the address of the NEW21 routine, and, once the Int 21h instruction is executed, our routine is placed in memory. From this point on, all calls by the system to Int 21h will first be processed by our NEW21 routine.

As stated elsewhere in this book, TRAPDEL also deallocates its environment block to save 256 bytes of memory. TRAPDEL does not access nor need a copy of the environment block and therefore it is discarded. Function 49h of Int 21h, Release Memory Block, is used to deallocate the memory reserved for the environment block. The segment address of the block of memory you want to free is loaded into ES and the AX register holds a count of the number of paragraphs of memory to be freed.

```
mov     ax,ds:[002ch]          ;environment block
mov     es,ax                  ;into ES now
mov     ah,49h                 ;release memory function
int     21h                    ;call dos
```

The final step taken in the initialization routine is to make TRAPDEL resident in memory and to reserve enough space for its code and data. This is done with the following code:

```
mov     dx,(offset end_prog - offset codesg +15) shr 4
mov     ax,3100h               ;terminate and stay resident
int     21h                    ;call dos
```

This exact same routine was used in TRAPBOOT, and you should refer to that chapter for a discussion of how Function 31h of Int 21h, Terminate and Stay Resident, is used to make a program resident in memory.

Fooling DOS

Once TRAPDEL is resident in memory, it monitors every DOS Int 21h function call. As has been stated earlier, there are two function calls that can be used to delete files from a disk. These are both intercepted, as shown below, in the NEW21 procedure.

```
new21   proc    far                     ;our new delete file routine
        assume ds:nothing,es:nothing
        pushf                           ;save flags on stack for now
us:     cmp     ah,13h                  ;is it request to delete file?
        je      del_fcb                 ;yes, do it then!
        cmp     ah,41h                  ;is it request to delete file?
        je      del_asc                 ;yes, do it then!
do_dos: popf                            ;no, recover flags from stack
        cli                             ;turn interrupts off now
        jmp     cs:old21h               ;do original DOS function
```

At the label US, the CMP instruction is used to compare the function code in the AH register with the function codes 13h and 41h, those of the delete file calls. If AH is not equal to one of these values, control passes through to the label DO_DOS, and the original Int 21h service is allowed to proceed in its normal fashion.

In the event that this Int 21h service indicates that a file is to be deleted, the program branches to the label DEL_FCB or to DEL_ASC, depending on which service has been requested.

Deleting FCB Files

If you will recall, in Chapter 2, we developed a program that renames directories. Through that program's development, we learned what a File Control Block is—a structure used by DOS that describes the file you want to manipulate. In addition, a program could also use an extended FCB to specify a file with a specific attribute which tells DOS what type of file it is (hidden, directory, etc.). TRAPDEL also uses the FCB method of file access.

Let's assume that Function 13h is being called to delete a file from the disk. When this function is called, it requires two parameters: AH must hold the function number (13h) and DS:DX must hold the segment and offset address of an FCB. As soon as the service has been executed, the file(s) identified by the FCB will be deleted. Note that you cannot delete files that are currently open or that have an attribute of read only.

However, if you fill the filename fields with a wild card character (either "*" or "?"), then Function 13h will remove all matching files from the disk.

Now, back to the routine starting at the label DEL_FCB, shown here:

```
        push    dx                      ;save DX
        mov     si,dx                   ;SI=pointer to FCB
        inc     si                      ;move past drive code first
        cld                             ;string moves go forward now
        call    fake_it                 ;and do it
        jc      fcb1                    ;go if error (no file found)
        pop     dx                      ;restore DX
        pop     ax                      ;restore AX
        popf                            ;recover flags first
        ret 2                           ;return to Int 21h calls with flags
fcb1:   pop     dx                      ;restore DX
        pop     ax                      ;restore AX
        popf                            ;recover flags first
        ret 2                           ;return to Int 21h with flags
```

When DOS or an application program uses Function 13h of Int 21h to delete a file, the AH register holds the function code and DS:DX points to the FCB. Therefore, the first action taken in the routine shown above is to temporarily save the contents of registers AX and DX on the stack. This is done because TRAPDEL needs to use these register values to move the file to the GARBAGE directory and then, after recovering AX and DX from the stack, to pass control back to DOS so it can actually remove the file from the disk. In this way, DOS is then fooled into believing the file was really removed when in fact it was simply moved to another directory.

Once these two registers have been preserved on the stack, the address of the FCB is transferred to SI. Next, SI is incremented by one to move to the second byte of the FCB, i.e., the starting address of the actual filename itself. The procedure FAKE_IT is then called to attempt to move the file to another directory.

If the FAKE_IT procedure was successful, the Carry Flag will be clear and the original values of DX and AX are popped off the stack, along with the Flags register. The interrupts are then turned off temporarily by issuing a CLI instruction, and control is returned to the calling application via the IRET instruction.

In the event that the FAKE_IT procedure was not able to move the file to the GARBAGE directory, the Carry Flag will be set. The code at the label FCB1 recovers AX and DX from the stack, as was just explained.

When you delete a file using the DOS DELete command, you may specify one individual file or several files if you use the wild card file specification. This means that TRAPDEL must also work when the wild card filename is specified. TRAPDEL uses the Find First Matching File and Find Next Matching File functions to make sure that all files have been processed. If we had not done this, then TRAPDEL would only process one file at a time and this would not, of course, be acceptable.

Deleting ASCIIZ Files

The newer DOS service that is used to delete files is Function 41h of Int 21h. On entry, AH holds the function code (41h) and DS:DX holds the address of an ASCIIZ file specification for the file to be deleted. Unlike the FCB delete file function, Function 41h can only delete one specific file at a time. The wild card "*" and "?" characters, therefore, cannot be used. On the other hand, this function can remove a file from any directory on the disk simply by including a pathname as part of the ASCIIZ file specification.

If Function 41h is successful, the Carry Flag is clear. If an error occurs and the function was unable to delete the file, the Carry Flag will be set and AX will hold an error code. An error will occur in situations where the file has a read-only attribute or if any element of the pathname does not exist.

In TRAPDEL, the DEL_ASC routine is called if DOS indicates that Function 41h should be used to delete the file. The DEL_ASC routine, shown here, sets CX to the maximum length of a pathname and SI to the address of the ASCIIZ pathname. Next, the FAKE_IT_ASC procedure is called.

```
del_asc: push   ax              ;save AX and DX
         push   dx              ;for now
         mov    cx,64d          ;maximum length for pathname
         mov    si,dx           ;SI=pointer to ASCIIZ pathname
         cld                    ;string moves go forward now
         call   fake_it_asc     ;show filename to be killed
         jc     asz_1           ;go if error (no such file)
         pop    dx
         pop    ax
         popf                   ;recover flags too
         ret 2                  ;return to Int 21h with Flags
asz_1:   pop    dx              ;recover DX
         pop    ax              ;and AX
```

```
        popf                            ;don't forget the flags
        ret 2                           ;return to Int 21h with Flags
new21   endp                            ;end of our routine
```

After TRAPDEL's own routine has moved the to-be-deleted file to the GARBAGE directory, the program branches to the label DO_DOS. Here, the Flags register is recovered from the stack and the interrupts are disabled. Control is then returned to the original interrupt routine that processes Int 21h function calls. This allows DOS to execute Function 41h as if TRAPDEL had not intercepted it in the first place.

Preparing to Move Files

The FAKE_IT procedure is responsible for retrieving the filename from the FCB when TRAPDEL detects a call to Function 13h. To this end, the procedure MK_FNAME is called to read each byte of the FCB filename and store it in the buffer FNAME as an ASCIIZ string.

```
mk_fname proc   near
        mov     cx,8d                   ;length of filename
        mov     di,offset fname         ;destination buffer
make_1: lodsb                           ;get one byte now
        cmp     al,' '                  ;is it a space?
        je      got_sp                  ;yes, process it
        stosb                           ;save character in FNAME
        loop    make_1                  ;loop until filename copied
dot:    mov     al,'.'                  ;get a dot separator
        stosb                           ;save character in FNAME
dot_1:  mov     cx,3d                   ;length of filename extension
make_2: lodsb                           ;get one byte
        stosb                           ;save character in FNAME
        loop    make_2                  ;loop until extension copied
        mov     al,0                    ;make this an ASCIIZ filename
        stosb                           ;store the zero byte
        ret                             ;return to caller
got_sp: lodsb                           ;get the next byte
        cmp     al,' '                  ;is it also a space?
        jne     no_sp                   ;no, so exit routine
        loop    got_sp                  ;yes, skip this space byte
        jmp     dot                     ;go get extension now
no_sp:  push    ax                      ;save original character
        mov     al,'.'                  ;get a period delimiter
        stosb                           ;save it
        pop     ax                      ;recover original character
        stosb                           ;save it too
        jmp     dot_1                   ;do the extension
mk_fname endp
```

This procedure looks a lot more complicated than it really is. The first nine lines of code set the DI register to the address of the FNAME buffer. Next, CX is set to a value of eight. This is the maximum length allowed by DOS for a filename. The program then goes into a simple loop routine which retrieves, via the LODSB instruction, one character from the FCB addressed by the SI register. This character is then compared to a space character. If the character is a space, the program branches to the label GOT_SP. This short section of code simply skips through the FCB filename until a non-space character is found in the buffer. Then the program branches back to the label DOIT, which inserts a period byte. (DOS filenames are eight characters, followed by a delimiting period byte, and then followed by a three-character filename extension). After the period delimiter is stored in the FNAME buffer, the filename extension is then copied. The string in the FNAME buffer is finally converted to an ASCIIZ string by appending a zero byte to the end of the filename and the procedure is terminated.

The next section of code in the FAKE_IT procedure calls the FND_FILE function to determine if the specified file exists on the disk, which will be discussed shortly.

Creating Directories

Before TRAPDEL can actually move a file to the GARBAGE directory, this special subdirectory must be created on the disk. The MK_DIR procedure shown here does this.

```
mk_dir  proc    near
        mov     ah,19h              ;get default disk function
        int     21h                 ;call dos (drive in AL 0=A, 1=B)
        add     al,'A'              ;convert to letter format
        mov     bx,offset dirname   ;directory to be created
        mov     [bx],al             ;store drive code in first byte
        mov     dx,offset dirname   ;directory to be created
        mov     ah,39h              ;create directory function
        int     21h                 ;call dos
        ret                         ;return to caller
mk_dir  endp
```

Since TRAPDEL can be called into action at any time, it must determine which disk drive is the one currently being used. Function 19h of Int 21h, Get Default Disk, is used to retrieve the drive code into the AL register. The drive code, where 0=A, 1=B, etc., is then converted to an ASCII character by adding a value of 65d to the value in AL. Next, this ASCII character is stored in the first byte of the buffer DIRNAME.

At the very beginning of the source code listing for TRAPDEL, you'll find the DIRNAME buffer defined as:

```
          DIRNAME     DB     'D:\GARBAGE',0
```

What we are doing now is simply inserting the ASCII drive letter into the DIRNAME buffer to replace the existing "D" letter, as shown above.

Once this step has been done, Function 39h of Int 21h is used to create the new directory. This function requires only two parameters: the function code (39h) is stored in AH and the address of the ASCIIZ pathname to be created is loaded into DS:DX.

If the function was successful, the Carry Flag will be clear, otherwise it will be set and an error code will be returned in AX. The MK_DIR procedure does not check for any errors, however; you may wish to add this code yourself. After the GARBAGE directory has been created, control returns to the calling routine.

Deleting Multiple Files

Another procedure is used by TRAPDEL to determine if one or more files should be moved to the GARBAGE directory. These two procedures are shown below.

```
fnd_file proc    near
         mov     dx,offset dta        ;disk transfer area buffer
         mov     ah,1ah               ;set DTA function
         int     21h                  ;call dos
         mov     dx,offset fname      ;file to search for (FCB)
         mov     cx,00h               ;file attribute (normal only)
         mov     ah,4eh               ;find first matching file
         int     21h                  ;call dos
         ret                          ;return to caller with CF set
fnd_file endp

fnd_next proc    near
         mov     dx,offset fname      ;file to search for (FCB)
         mov     cx,00h               ;file attribute (normal only)
         mov     ah,4fh               ;find next matching file
         int     21h                  ;call dos
         ret                          ;return to caller
fnd_next endp
```

The first procedure to be called is FND_FILE. To begin with, the Disk Transfer Address is set to TRAPDEL's own buffer called DTA. This DTA buffer will hold information about each file in turn that is found by the Find File function calls.

The actual file search routine is then invoked. DX holds the address of the target file(s) to be looked for and the Find First Matching File function is called. At this point, only normal files will be searched for (the file attribute is set to 0 in CX).

If a matching file was found on the disk, then TRAPDEL moves this file to the GARBAGE directory. Then the FND_NEXT procedure is called to locate the next possible matching wild card filename. This process is repeated until all files have been moved to the GARBAGE directory.

Summary

In examining the code in TRAPDEL.ASM, a number of techniques were presented for creating terminate and stay resident programs. These techniques include:

- monitoring Int 21h function calls for specific service calls
- changing and modifying interrupt service routines
- shrinking memory required by the TSR to a minimum in order to preserve the most memory possible for other TSR utilities and application programs

Projects

1. When TRAPDEL is installed, the deleted files are moved to the GARBAGE directory on the default disk. Allow a different subdirectory to be created. This could be done with a command line parameter such as: TRAPDEL /D=C:\JUNK.
2. Add more complete error-checking routines to various parts of the program.

Program Listing

```
;TRAPDEL.ASM
;<c> 1992 Deborah L. Cooper
;
;TRAPDEL intercepts DOS's file delete function and places the
;to-be-deleted file in a special GARBAGE directory on the same
;disk drive.
;
codesg  segment                         ;start of CODE segment
        assume  cs:codesg,ds:codesg,es:codesg
        org     100h                    ;make this a COM program
start:  jmp     init                    ;go make our program resident

author          db      'TRAPDEL <c> 1992 by Deborah L.'
                db      'Cooper',0dh,0ah,'$'
bad_alloc       db      'Memory allocation error - aborting','$'
msg8            db      'Unable to move file','$'
dirname         db      'D:\GARBAGE',0 ;storage directory name
nofile          db      'File does not exist','$'
old21h          dd      ?               ;original DOS interrupt vector
flag            db      0               ;1=to be installed now
other_seg       dw      0               ;segment of installed TRAPDEL
fname           db      63 dup(0)       ;filename to be 'deleted'
fname2          db      63 dup(0)       ;ASCIIZ filename
movname         db      63 dup(0)       ;D:\GARBAGE\filename,0
dta             db      128 dup(0)      ;disk transfer buffer
;=============================================================
;Every time DOS or an application wants to delete a file, we come
;here and do our own routine instead
;=============================================================
new21   proc    far                     ;our new delete file routine
        assume  ds:nothing,es:nothing
        pushf                           ;save flags on stack for now
us:     cmp     ah,13h                  ;is it request to delete file?
        je      del_fcb                 ;yes, sound a bell then!
        cmp     ah,41h                  ;is it request to delete file?
        je      del_asc                 ;yes, show filename now
do_dos: popf                            ;recover the flags from stack
        cli                             ;turn interrupts off now
        jmp     cs:old21h               ;do the original DOS
function/exit
;=============================================================
;This routine "deletes" the file using the File Control
;Block delete file function
;=============================================================
del_fcb:push    ax                      ;save AX
        push    dx                      ;save DX
        mov     si,dx                   ;SI=pointer to File Control Block
        inc     si                      ;move past drive code first
        cld                             ;string moves go forward now
        call    fake_it                 ;and do it
        jc      fcb1                    ;go if error (no file found)
        pop     dx                      ;restore DX
        pop     ax                    ;restore AX
;Now we want to fool DOS into thinking it actually deleted this
;particular file.
        popf                          ;recover flags first
        ret 2                         ;return with Flags
fcbl:   pop     dx                    ;restore DX
        pop     ax                    ;restore AX
        popf                          ;recover flags first
        ret 2                         ;return to Int 21h with Flags
;=========================================================
;This routine "deletes" the file using the ASCIIZ string
;delete file function
;=========================================================
del_asc:push    ax                    ;save AX and DX
        push    dx                    ;for now
        mov     cx,64d                ;maximum length for pathname
        mov     si,dx                 ;SI=pointer to ASCIIZ pathname
        cld                           ;string moves go forward now
        call    fake_it_asc           ;show filename to be killed first
        jc      asz_1                 ;go if error (no such file)
        pop     dx
        pop     ax
        popf                          ;recover flags too
        ret 2                         ;return to Int 21h with Flags now
asz_1:  pop     dx                    ;recover DX
        pop     ax                    ;and AX
        popf                          ;don't forget the flags
        ret 2                         ;return to Int 21h with Flags now
new21   endp                          ;end of our routine

;This routine moves the file to the GARBAGE directory on the
;default disk drive
;On entry:  DX=ASCIIZ filename to be 'deleted'
;               or DS:DX points to File Control Block +1 byte
;           AH = 41h delete file function code

fake_it proc    near
        push    cs                    ;we need to have access to
        pop     es                    ;TRAPDEL's data segment
        assume  es:codesg             ;tell the system now
        call    mk_fname              ;get filename from FCB format
;We must save the segment registers prior to do the rest of
;these functions as DOS requires them to be returned as they
;were before we interrupted DOS itself.
        push    ds                    ;save data segment
        push    es                    ;save extra segment

        push    cs                    ;need to access our own
        pop     ds                    ;code and data segments
        push    cs
        pop     es
        assume  cs:codesg,ds:codesg,es:codesg  ;tell the system!
        call    fnd_file              ;does file(s) to delete exist?
        call    mk_dir                ;create the GARBAGE directory
```

```
do_file:call    mk_name                 ;append DIRNAME & FNAME together
;Now we are prepared to move the file to the GARBAGE directory
        mov     dx,offset dta+30d       ;file to be moved
        mov     di,offset movname       ;destination dir and file names
        mov     ah,56h                  ;move/rename file function
        int     21h                     ;call dos
;If we have more than one wildcard file to process, go back for
;next one!
        jmp     next_one                ;go process next wildcard
filename
;What do we do if there is an error when we attempt to move file?
fake2:  stc                             ;show there is an error (no file)
        pop     ax                      ;restore segments
        mov     es,ax
        pop     ax
        mov     ds,ax
        assume  ds:nothing,es:nothing
        ret                             ;return to caller
fake1:  pop     ax                      ;restore extra segment
        mov     es,ax
        pop     ax                      ;restore data segment
        mov     ds,ax
        assume  es:nothing,ds:nothing   ;tell the system
        ret                             ;return to caller
next_one:
        call    fnd_next                ;find next file
        jc      fake1                   ;exit when all files processed
        jmp     do_file                 ;and continue
fake_it endp
;==============================================================
;This routine sets us up to delete an ASCIIZ file(s)
;==============================================================
fake_it_asc     proc    near
        push    cs                      ;we need to have access to
        pop     es                      ;TRAPDEL's data segment
        assume es:codesg                ;tell the system now
        cld                             ;copy bytes left to right
        mov     si,dx                   ;SI=address of ASCIIZ pathname
        mov     di,offset fname         ;destination buffer
        mov     cx,64d                  ;max possible length of pathname
        rep movsb                       ;copy pathname to FNAME buffer

;We must save the segment registers before we do anything else so
;that they can be restored for DOS to carry on with its work.
        push    ds                      ;save data segment
        push    es                      ;save extra segment
        push    cs                      ;we need to access our
        pop     ds                      ;own code and data segments
        push    cs
        pop     es
        assume cs:codesg,ds:codesg,es:codesg
        call    mk_dir                  ;create the GARBAGE directory
        call    fnd_file                ;does the file exist?
```

```
;Insert code here to strip filename from FNAME buffer
do_asc: mov     di,offset fname         ;point to pathname
        mov     cx,0ffffh               ;maximum possible length
        call    strlen                  ;calculate length of pathname
;Now find the start of the filename by going backwards and
;searching for the first slash (/) byte.
        mov     di,offset fname         ;point to pathname
        add     di,cx                   ;CX=length of pathname
        call    findsl                  ;find the slash byte
        inc     di                      ;don't copy the slash though!
        inc     di                      ;to our buffer
        xchg    si,di                   ;SI=text to copy out
        mov     di,offset fname2        ;destination buffer
        mov     cx,13d                  ;copy this many bytes max
        cld                             ;copy left to right
        rep movsb                       ;copy it now
;Now copy the ASCIIZ filename back to the FNAME buffer so we
;can use the subroutines we've already debugged!
        mov     si,offset fname2         ;source filename
        mov     di,offset fname         ;destination buffer
        mov     cx,13d                  ;maximum ASCIIZ filename length
        cld                             ;copy bytes left to right
        rep movsb                       ;copy ASCIIZ filename to FNAME
        call    mk_name                 ;append GARBAGE + ASCIIZ filename

;Move the file to the GARBAGE directory
        mov     dx,offset dta+30d       ;ASCIIZ filename to be moved
        mov     di,offset movname       ;destination dir & file names
        mov     ah,56h                  ;move/rename file function
        int     21h                     ;call dos

next_asc:call   fnd_next                ;see if another file to do
        jc      asz_3                   ;no, so just exit if done
        jmp     do_asc                  ;else go back to do next file

asz_3:  pop     ax                      ;restore extra segment
        mov     es,ax
        pop     ax                      ;restore data segment
        mov     ds,ax
        assume  ds:nothing,es:nothing   ;tell the system now
        ret                             ;return to caller
fake_it_asc     endp

;====================================================================
;This routine determines whether the requested file(s) to delete
;actually exists.
;====================================================================
fnd_file        proc    near
        mov     dx,offset dta           ;disk transfer area buffer
        mov     ah,1ah                  ;set DTA function
        int     21h                     ;call dos
        mov     dx,offset fname         ;file to search for (FCB)
        mov     cx,00h                  ;file attribute (normal only)
        mov     ah,4eh                  ;find first matching file
        int     21h                     ;call dos
        ret                             ;no, return with CF set
fnd_file        endp

fnd_next        proc    near
        mov     dx,offset fname         ;file to search for (FCB)
        mov     cx,00h                  ;file attribute (normal only)
        mov     ah,4fh                  ;find next matching file
        int     21h                     ;call dos
        ret                             ;return to caller
fnd_next        endp
;=============================================================
;This routine reads the filename from the File Control Block
;and saves it to our buffer FNAME
;=============================================================
mk_fname        proc    near
        mov     cx,8d                   ;length of filename
        mov     di,offset fname         ;destination buffer
make_1: lodsb                           ;get one byte now
        cmp     al,' '                  ;is it a space?
        je      got_sp                  ;yes, process it
        stosb                           ;save character in FNAME
        loop    make_1                  ;loop until filename copied
dot:    mov     al,'.'                  ;get a dot separator
        stosb                           ;save character in FNAME
dot_1:  mov     cx,3d                   ;length of filename extension
make_2: lodsb                           ;get one byte
        stosb                           ;save character in FNAME
        loop    make_2                  ;loop until extension copied
        mov     al,0                    ;make this an ASCIIZ filename
        stosb                           ;store the zero byte
        ret                             ;return to caller
got_sp: lodsb                           ;get the next byte
        cmp     al,' '                  ;is it also a space?
        jne     no_sp                   ;no, so exit routine
        loop    got_sp                  ;yes, skip this space byte
        jmp     dot                     ;go get extension now
no_sp:  push    ax
        mov     al,'.'
        stosb                           ;save character first!
        pop     ax
        stosb
        jmp     dot_1
mk_fname        endp
;=============================================================
;This routine attempts to create the GARBAGE directory if
;it does not already exist
;=============================================================
mk_dir  proc    near
        mov     ah,19h                  ;get default disk function
        int     21h                     ;call dos (drive in AL 0=A,1=B)
        add     al,'A'                  ;convert to letter format
        mov     bx,offset dirname       ;directory to be created
        mov     [bx],al                 ;store drive code in first byte
        mov     dx,offset dirname       ;directory to be created
        mov     ah,39h                  ;create directory function
        int     21h                     ;call dos
        ret                             ;return to caller
mk_dir  endp
;
;=========================================================================
;This routine creates a fully qualified pathname from the
;GARBAGE + filename strings in DIRNAME and FNAME
;=========================================================================
mk_name proc    near
        mov     si,offset dirname       ;directory name (GARBAGE)
        mov     di,offset movname       ;destination buffer
        mov     cx,63d                  ;maximum length for a pathname
        cld                             ;clear direction flag first
name1:  lodsb                           ;get one character
        cmp     al,0                    ;reached end of dirname?
        je      name2                   ;yes, go
        stosb                           ;no, store the byte
        loop    name1                   ;go back for another
;Now insert a slash '\' between directory name and filename
name2:  mov     al,'\'                  ;get a slash character
        stosb                           ;store in MOVNAME buffer
;Now append filename to MOVNAME buffer
        mov     si,offset dta           ;fname ;filename to be copied out
        add     si,30d
        mov     cx,13d                  ;max length of filename
name3:  lodsb                           ;get one character
        cmp     al,0                    ;end of filename reached?
        je      name4                   ;yes, process it as ASCIIZ then
        stosb                           ;no, store the character
        loop    name3                   ;go back for another
;Make MOVNAME into an ASCIIZ pathname now
name4:  mov     al,0                    ;get a zero byte
        stosb                           ;store it at end of MOVNAME
        clc
        ret                             ;return to caller
mk_name endp
;===================================================================
;This routine calculates the length of an ASCIIZ string
;===================================================================
strlen  proc    near
        xor     al,al                   ;search for a zero byte
        cld                             ;search left to right
        repnz scasb                     ;look for the zero now
        not     cx                      ;ones complement of CX
        dec     cx                      ;CX=length of string
        ret                             ;return to caller
strlen  endp
;===================================================================
;This routine finds the start of the filename
;===================================================================
findsl  proc    near
        mov     al,'\'                  ;search for a slash byte
        std                       ;search right to left
        repne scasb               ;look for the slash byte now
        jcxz   no_slash           ;we couldn't find one!
        ret                       ;return to caller
no_slash:
        ret
findsl  endp
;===================================================
;The following code makes TRAPDEL resident in memory
;===================================================
end_prog      =     $             ;marks end of resident code

msg3    db     'TRAPDEL is already installed','$'
msg5    db     'TRAPDEL has been removed from memory now','$'
msg6    db     'Unable to remove TRAPDEL from memory','$'

init    proc   near
        assume cs:codesg,ds:codesg
        cld                       ;clear DF first
        not    word ptr start     ;destroy our first word's data
        xor    bx,bx              ;search from first segment
        mov    ax,cs              ;compare to this code segment

next:   inc    bx                 ;do this segment now
        cmp    ax,bx              ;until reaching our own
        mov    es,bx              ;code segment
        je     notyet             ;not installed yet
        mov    si,offset start    ;setup to compare the
        mov    di,si              ;strings at DI and SI
        mov    cx,16              ;compare this many bytes
        rep cmpsb                 ;do it now
        or     cx,cx              ;did the strings match?
        jnz    next               ;no, try next segment then
        mov    other_seg,es       ;yes, save TRAPDEL's code segment
        jmp    not_in             ;go check for commands
;Set flag because TRAPDEL is not resident yet, but we have
;been instructed to make it resident
notyet: mov    flag,1             ;set flag to install later

;Now check the command line for possible switches
not_in: mov    si,80h             ;address of command line
        lodsb                     ;get length of it in AL
        cmp    al,0               ;are there any switches?
        jz     make_r             ;no, go install then

com_2:  lodsb                     ;get next byte from line
        cmp    al,0dh             ;is it a carriage return?
        je     make_r             ;yes, exit this loop then
        cmp    al,20h             ;is it a space?
        je     com_2              ;yes, skip all leading spaces
        cmp    al,'/'             ;is it a slashbar?
        jne    make_r             ;no, go install now
        lodsb                     ;yes, get next byte
        and    al,5fh             ;convert to uppercase
        cmp     al,'U'                ;want to un-install now?
        jne     make_r                ;no, go install it
        jmp     uninstall             ;yes, go remove it

;Set interrupt vectors to our routine
make_r: cmp     flag,1                ;install it now?
        je      put_r                 ;yes, go do it
        mov     dx,offset msg3        ;no, already here message
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        mov     ah,4ch                ;terminate program function
        int     21h                   ;call dos

put_r:  mov     ah,35h                ;get interrupt vector function
        mov     al,21h                ;for DOS functions
        int     21h                   ;call dos
        mov     word ptr old21h,bx    ;save offset address
        mov     word ptr old21h[2],es ;save segment address
        mov     ah,25h                ;set interrupt vector function
        mov     al,21h                ;for DOS functions
        mov     dx,offset new21       ;to our NEW21 routine
        int     21h                   ;call dos

;Display copyright notice
        mov     dx,offset author      ;copyright notice
        mov     ah,09h                ;display string function
        int     21h                   ;call dos

;Release environment to save some memory
        mov     ax,ds:[002ch]         ;environment block
        mov     es,ax                 ;into ES now
        mov     ah,49h                ;release memory function
        int     21h                   ;call dos

;Terminate and remain in memory
        mov     dx,(offset end_prog - offset codesg + 15) shr 4
        mov     ax,3100h              ;terminate program function
        int     21h                   ;call dos

;Attempt to remove TRAPDEL from memory
uninstall:
        assume  ds:codesg             ;set DS to code segment
        push    es                    ;save ES first
        mov     ah,35h                ;get interrupt vector function
        mov     al,21h                ;for DOS functions
        int     21h                   ;call dos
        mov     ax,es                 ;compare the address
        cmp     ax,other_seg          ;to TRAPDEL's
        jne     un_1                  ;exit - cannot remove it!

;Release memory occupied by TRAPDEL
        mov     ah,49h                ;release memory function
        int     21h                   ;call dos
        jc      un_1                  ;exit if error
        push    ds
        assume  ds:nothing
        lds     dx,es:[old21h]          ;DX=DOS interrupt address
        mov     ah,25h                  ;set interrupt vector function
        mov     al,21h                  ;for DOS functions
        int     21h                     ;call dos
        pop     ds                      ;recover DS
        assume  ds:codesg               ;to code segment
        not     word ptr es:[start]     ;erase our data here
        push    cs                      ;set DS to our
        pop     ds                      ;code segment
        assume  ds:codesg               ;tell the assembler

        mov     dx,offset msg5          ;removed message

noway:  mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        pop     es                      ;restore ES too
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos

un_1:   mov     dx,offset msg6          ;cannot remove message
        jmp     short noway             ;exit

init    endp                            ;end of initialization routine
codesg  ends                            ;end of code segment
        end     start                   ;end of program
```

Chapter 7

SAFE

Use SAFE to disable the DOS "FORMAT" command. It provides an example of using undocumented DOS function calls and further shows how to add new DOS commands or modify existing ones.

There is one controversy among programmers that never seems to be resolved. You'll either agree or disagree, but if you want to write robust, well-behaving memory resident software, you'll have to use some undocumented features of DOS. Many programmers shy away from using undocumented functions in their programs, claiming that Microsoft may change the way these functions work or remove them altogether from the operating system. They do have a valid point, but to this date, thousands of programmers have used these service calls in their programs and they have worked as advertised.

SAFE is a program that uses two undocumented functions available since DOS 3.3. Without these two service calls, adding new commands to the operating system would be considerably more difficult and time-consuming. This is why SAFE was developed. Through SAFE, you'll learn how to use undocumented AE00h and AE01h functions of Int 2Fh to disable the DOS "FORMAT" command. SAFE will also show you another type of terminate and stay resident program. It is not activated through keyboard action, like TRAPBOOT, nor by monitoring Int 21h functions, as in TRAPDEL. Instead, SAFE is activated whenever it detects the command "FORMAT" at the DOS prompt. In short, SAFE is a TSR that is event-driven.

Although the various sections of SAFE relating to the design of memory resident programs has been covered in previous chapters, you'll find there are a number of interesting things to be learned in this chapter. First, the routine used to intercept Int 2Fh calls is unique to SAFE and it should be studied carefully since it may give you ideas for other programs that might need to intercept DOS commands. Second, SAFE will demonstrate how you can successfully manipulate DOS through the previously mentioned undocumented Int 2Fh function calls.

Disabling DOS Commands on the Fly

Reloading software and datafiles is a bothersome, time-consuming chore none of us likes to do—all because you or someone else accidentally issued a DOS "FORMAT" command!

This is where SAFE comes to the rescue. SAFE is a non-popup terminate but stay resident program that disables DOS's formidable "FORMAT" command. SAFE can also easily be modified to disable any other DOS command you like, or, if you're really ambitious, you can use SAFE as a model to add new commands to the operating system.

SAFE prevents this particular DOS command from executing in its usual manner by intercepting all calls to Int 2Fh, the Multiplex Interrupt. In order to do this, SAFE uses two undocumented DOS function calls, AE00h and AE01h, of Int 2Fh. These two calls allow SAFE to change the way the operating system behaves. The only drawback is that this program will only work on computer systems running MS-DOS 3.3 or higher.

Functions Used in SAFE.ASM

Int 21h, AH=02h	Display byte
Int 21h, AH=09h	Display string
Int 21h, AH=25h	Set interrupt vector
Int 21h, AH=30h	Get DOS version
Int 21h, AH=35h	Get interrupt vector
Int 21h, AH=49h	Release block of memory
Int 21h, AH=4ch	Terminate process with return code
Int 21h, AX=3100h	Terminate and stay resident
Int 2Fh, AX=AE00h	Check for installed DOS command
Int 2Fh, AX=AE01h	Execute installed DOS command

How to Use SAFE

To run SAFE, simply type its name at the DOS prompt or put it in your AUTOEXEC.BAT file so it is installed in memory whenever you boot your computer system. SAFE needs less than 1K of RAM and will not interfere with other TSR programs.

Once SAFE has been installed in memory, you can issue the DOS "FORMAT" command. However, you will only receive a "I can't let you do that!" message for your trouble. When it is active in memory, SAFE will not allow DOS to format a floppy or hard disk. If you really do want to initialize a disk, then you must uninstall SAFE before issuing the "FORMAT" command. To do this, type the following at the DOS prompt: SAFE /U.

How SAFE Works

Just as there are two types of DOS programs, namely COM and EXE, so there are two kinds of TSR programs. The most familiar of these is activated when a hot-key combination is pressed on the keyboard. The other type of TSR is called a non-popup utility because it waits to be activated by a certain event (event processing). SAFE is such a program—it waits for the user to enter a specific DOS command before it jumps into action.

The two undocumented DOS function calls AE00h and AE01h make it possible for SAFE to check the command entered on the command line before COMMAND.COM has a change to process it. In this case, if the command is "FORMAT," then SAFE simply displays an error message and tells COMMAND.COM to ignore the request. All other DOS commands function normally.

Preparing to Work

The section of code that actually puts SAFE into memory is called the initialization (more correctly called the transient) section of a TSR program. Therefore, the first section of code executed by SAFE starts at the label INIT, which is located toward the end of the source code listing.

Determining the DOS Version

Because SAFE uses two undocumented DOS function calls that are only available under DOS 3.3 and later, SAFE's first task is to check the version of MSDOS currently installed in the computer system. This is done with Function 30h of Int 21h, as shown below:

```
mov     ah,30h              ;get DOS version function
mov     al,00h              ;to check
int     21h                 ;call dos
```

The major version number is returned in the AL register and the minor version number is returned in the AH register. Therefore, the first check SAFE makes is to ascertain if the major version number is 3 or greater. If it is 4 or greater, the program jumps to the label CHECK_2 to install the TSR in memory.

However, if the major version is 3, then SAFE must make sure the minor version number is at least .30. This is why a second check is made.

```
cmp     al,3                ;is it 3 or higher?
ja      check_2             ;go if 4 or greater
jb      no_dos              ;if less than 3, no go!
cmp     ah,01Eh             ;is minor version .30?
jae     check_2             ;no, can't do it then
```

If the computer system is running a DOS version less than 3.3, then the code at the label NO_DOS is executed. The DX register points to the address of the error message we want to display and Function 09h of Int 21h is called to output the string to the screen. Then the program is terminated by issuing an Int 21h, Function 4Ch call.

```
mov     dx,offset errmsg    ;point to error message
mov     ah,09h              ;display string function
int     21h                 ;call dos
mov     ah,4ch              ;terminate program function
int     21h                 ;call dos
```

If SAFE finds that the operating system fits its requirements, the program continues execution at the label CHECK_2. This section of the program attempts to determine if SAFE has been previously installed in memory. This prevents you from installing multiple copies of the same program in your system, and therefore, saves precious memory space. All TSR programs should include this feature automatically.

Changing Interrupt Vectors

The routine starting at the label CHECK_6 is responsible for making SAFE resident in memory. After the copyright notice is sent to the display, SAFE's own procedure for processing Int 2Fh calls must be inserted into the DOS Multiplex Interrupt vector.

```
mov     ah,35h                  ;get interrupt 2Fh vector
mov     al,2Fh                  ;2Fh code
int     21h                     ;call dos
mov word ptr [cs:old2Fh],bx     ;save offset and segment
mov word ptr [cs:old2Fh+2],es
mov     ah,25h                  ;point to new routine
lea     dx,header               ;to install
int     21h                     ;call dos
```

The first five lines of code above perform a necessary procedure that is standard when writing TSR programs. The address of the old service routine that processes Int 2Fh calls is first saved. This is done so that the routine that removes SAFE from memory can restore the Multiplex Interrupt as it was before we installed SAFE in the computer system. Function 35h of Int 21h, Get Interrupt Vector, is used to retrieve the interrupt vector's segment and offset address. This is returned in ES:BX, and we then save this address in the variable OLD2Fh for later processing.

Function 25h of Int 21h is then used to set the Multiplex Interrupt routine to SAFE's own procedure. The DS:DX register pair is loaded with the address of the HEADER routine and, once the Int 21h instruction is executed, our routine is placed in memory. From this point on, all calls by the system to Int 2Fh will first be processed by our HEADER routine.

Making it Resident in Memory

SAFE uses Function 31h to reserve the memory it needs. This is accomplished with the following three lines of source code:

```
mov     dx,(offset lastbyte - offset codesg + 15) shr 4
mov     ax,3100h
int     21h
```

We used Function 31h of Int 21h to make TRAPBOOT resident in memory. This very same technique is also used in our SAFE program. If you need to refresh your memory on how Function 31h works, you should refer to its description in the chapter on TRAPBOOT.

Event Processing the Easy Way

As was mentioned earlier in this section, SAFE is a special type of TSR program in that it works in the background, patiently waiting for a specific request from the operating system. This request is triggered whenever you type a command at the DOS prompt. When this is detected by SAFE, the HEADER routine, now stored in memory, jumps into action. Most other types of TSR programs are triggered by pressing a hot-key combination.

It follows then, that the first task of the HEADER routine is to determine if DOS has received a request via Function AE00h of Int 2Fh. This function ascertains whether or not the specified command typed by the user is a DOS command. In short, this function enables you to install non-popup extensions of sub-programs that DOS thinks are part of its own repertoire of commands. All that is needed is a method of determining if DOS is currently executing this undocumented function. This is done at the label OK_STAY, shown below.

```
ok_stay: cmp     ah,0aeh              ;look for our calling
         jnz     old_2f               ;not us, let COMMAND.COM finish
         cmp     al,01h               ;is it our calling?
         jz      reply                ;yes, do our routine then
```

In order to understand the above code, it is necessary to first describe how undocumented DOS functions AE00h and AE01h of Int 2Fh work.

A Little History

Whenever DOS is about to execute a command entered on the command line, it makes a call to the Multiplex Interrupt 2Fh. DOS sets the AX register to AE00h with DS:BX pointing to the command line buffer. This buffer consists of a one-byte value which tells DOS how long the command is, followed by the text of the command itself. In addition, DOS sets DS:SI to a buffer in the same format as that pointed to by DS:BX, except that all spaces have been removed from the command line, the command line has been converted to uppercase characters, and all trailing parameters have been removed.

As soon as DOS issues the Int 2Fh call to the routine handling this function, it immediately returns with a value in AL. If AL equals zero, then DOS knows it must execute the command in the usual way. On the other hand, if AL equals FFh, then DOS issues another Int 2Fh call, this time with AX=AE01h. As with the call to Function AE00h, DOS again sets DS:SI to point to the text entered on the command line. The first byte of this buffer also holds a value indicating the command's length.

This length byte is, in fact, the key to SAFE's ability to tell DOS not to execute the FORMAT command. If this value is set to zero, DOS will not execute the command pointed to by DS:SI. Instead, DOS thinks that the new Int 2Fh handler has already executed the command itself. This is done in SAFE's HEADER routine.

Obviously the idea behind the HEADER routine, then, is to trap all calls to Int 2Fh. Since DOS always calls Function AE00h first, we check it with the following code shown below:

```
ok_stay: cmp     ah,0aeh              ;look for our calling
         jnz     old_2f               ;not us, let COMMAND.COM finish
         cmp     al,01h               ;is it our calling?
         jz      reply                ;yes, do our routine then
```

If Int 2Fh is called with AH equal to a value other than AEh, then the original Int 2Fh service routine is called. Otherwise, the second CMP instruction determines if AX=AE01h, signaling that DOS is about to execute the command entered on the command line.

The section of code starting at the label REPLY checks for two possible avenues that can be taken by SAFE. If the variable REQUEST is set to a value of one, then we know that SAFE was asked to remove itself from memory. The count byte is also set to 0 to tell DOS that this command is assumed to have been executed by the Int 2Fh handler, in this case our HEADER routine. The next step is for the REMOVE procedure to perform the steps necessary to remove our Int 2Fh HEADER routine from memory, thereby allowing DOS to process the FORMAT command in the usual manner.

```
reply:   pushf                        ;save the flags
         mov     es:word ptr [si],0   ;set count byte to zero to
                                      ;tell COMMAND.COM we will process
                                      ;this command ourselves
         cmp     requst,1             ;are we to terminate?
         jnz     do_format            ;if not, go do the other thing
         popf                         ;recover the flags (not needed)
         call    remove               ;uninstall SAFE now
         mov     dx,offset msg2       ;show uninstalled message
         jc      er_exit              ;can't do it for some reason
         mov     dx,offset msg3       ;was un-installed ok message
         mov     ah,09h               ;display string function
         int     21h                  ;call dos
         iret                         ;all done and gone!
```

```
er_exit: mov    ah,09h                  ;display string function
         int    21h                     ;call dos
         mov    ah,4ch                  ;terminate program function
         int    21h                     ;call dos
```

For the time being, let's assume that the command entered was not a request to uninstall SAFE from memory. In this case, the program would branch to the label DO_FORMAT.

We don't want DOS to execute its own "FORMAT" command; instead, we want to display the message "I can't let you do that!" and just return without further processing. The code fragment shown below uses DOS Function 02h of Int 21h, Display Byte, to display each character in the string MSG.

```
do_format:
         mov    bx,offset cs:msg        ;get our response
         mov    ah,02h                  ;display byte function
show:    mov    dl,cs:byte ptr [bx]     ;get one byte now
         cmp    dl,024h                 ;have we reached end of MSG?
         je     done                    ;yes, then exit
         int    21h                     ;no, display this byte
         inc    bx                      ;bump buffer pointer
         jmp    show                    ;do the next
done:    mov    al,00h                  ;report our status
         popf                           ;reset the flags
         iret                           ;back to command.com
```

Not only is the above routine showing us how to display a string of characters (terminated by a dollar sign ($) byte) using Function 02h of Int 21h, but it also shows an interesting technique—saving execution time. Notice that the function code is only set once outside the SHOW routine. Since the contents of the AH register are not modified in any way in this routine, it need only be set once. This saves time by not having to execute the statement MOV AH,02h each time a new character is retrieved from the buffer pointed to by BX.

At the label DONE, the AL register is set to zero. This notifies DOS that this command has already been processed by the HEADER service routine for Int 2Fh. By setting AL equal to zero, DOS knows it does not need to execute the command itself, since its already been done through and by the new interrupt 2Fh handler.

Comparing Strings

Now, let's go back to the code, starting at the label ISUS. If the previous CMP instruction finds that a call was indeed made by DOS with AX equal to AE00h, then the next step is to determine if the command entered on the command line was "FORMAT." Here's how this is done:

```
isus:   push    si                      ;save their stuff
        push    cx                      ;for later
        mov     ch,0                    ;get command size in CX
        mov     cl,byte ptr [si]        ;from the buffer
        inc     si                      ;move up to text of command
        push    bx                      ;save their stuff
        mov     bx,offset cs:command    ;point to our command
check:  mov     al,byte ptr [si]        ;are the two chars the
        inc     si                      ;same in both buffers?
        cmp     al,cs:[bx]              ;check ours
        jnz     no_com                  ;if not, let COMMAND.COM
                                        ;process this command
        inc     bx                      ;move up to next byte to check
        loop    check                   ;check the whole word
        mov     ax,0aeffh               ;report our status 'its ours'
        jmp     go_com                  ;tell COMMAND.COM we are here
```

This routine is used to see if the "FORMAT" command is the command DOS is about to execute. As stated earlier, when DOS sets AX equal to AE00h, it also sets DS:BX to point to the command line buffer, with the first byte set to the length of this command. DS:SI also points to this information, but with the text of the command line stripped off and all unnecessary space characters removed and the command stored in uppercase characters.

Knowing this, all SAFE has to do at this point is to retrieve the length of DOS's command in CX and compare the string at DS:SI with our "FORMAT" string pointed to by BX. If the two command strings match, the statement MOV AX,0AEFFh is executed to tell DOS to issue another call to Int 2Fh's Function AE01h to process the command, which eventually goes to the DO_FORMAT routine in SAFE and doesn't allow the "FORMAT" command to execute.

If the two strings do not match, then the program branches to the label NO_COM. The code fragment here simply restores the previously saved registers to their original status by POPing them off the stack and returns control to the original Int 2Fh handler. The command entered on the command line is then executed by DOS without any more intervention from SAFE.

```
no_com:  mov     ax,0ae00h           ;say it's ours
go_com:  pop     bx                  ;restore all the registers
         pop     cx                  ;as they were before
         pop     si                  ;we interrupted this call
         popf                        ;get their flags back too
         jmp     old_2f              ;and do original handler
```

The statement JMP OLD_2F used in the above code tells the system to branch to the address contained in the variable OLD_2F. If you will recall, this routine's address was saved when SAFE installed itself in memory. Therefore, the address stored in OLD_2F is the address of the original Int 2Fh function and we are simply doing a far jump to this memory location.

Uninstalling Memory Resident Programs

Removing a TSR program from memory is a very simple task if done in a methodical manner. It cannot be stressed enough that a TSR program must be the last program loaded into memory before it can be removed. If this precaution is not adhered to, the memory blocks within the PC will become fragmented, requiring that the computer system be rebooted. In addition, more than one TSR program may have grabbed the same interrupt vector used by a different TSR utility.

You can use several techniques to determine if your TSR was the last one to be made memory resident. However, the most common method is to compare the current address of an interrupt vector with the address of the routine your TSR uses to service the interrupt. If these two addresses are identical, then you can safely assume no other TSR has been loaded into memory after yours.

To remove a TSR program from memory successfully, several steps need to be taken. These steps do not have to always be performed in this exact order—that will depend on your particular preference and on your program's purpose.

First, all modified interrupt vectors must be restored to their original state. For example, if you hook into the Int 21h interrupt vector, then the original routine that processes Int 21h calls must be restored. In many TSRs, more than one interrupt vector is hooked and you must remember to restore all those used by your program.

Second, if your TSR program used any datafiles, then these must be closed prior to removing the program from memory. Closing all opened files will ensure that data stored in the I/O buffers is physically written to the disk file and that all file handles will be released back to DOS for future use.

Third, the memory reserved for the TSR program must be deallocated back to the system pool. This block of memory can then be used by other TSR programs or by DOS applications.

SAFE takes these steps to remove itself from memory and can be used as a model to create other TSR programs with this uninstall feature.

Removing SAFE from Memory

Like the TRAPDEL program discussed earlier in this book, SAFE includes an option of removing itself from memory. If the "/U" parameter is found on the DOS command line, SAFE will attempt to remove itself from memory immediately. Chapter 5 discusses in great detail how to retrieve switches and parameters from the command line. This same technique is used in SAFE to detect the "/U" switch.

If the uninstall option was found on the command line, then the procedure REMOVE gets executed. The first step SAFE must take is to make sure that the Int 2Fh vector has not been modified by another program since SAFE was installed. The only way to determine this is to use Function 35h of Int 21h to retrieve the segment and offset address of the current service routine for Int 2Fh. This information will be returned in ES:BX. All SAFE has to do is load the address of the HEADER routine into CX and then compare the two addresses, as shown here:

```
remove  proc    near
        push    es
        mov     cx,offset header        ;point to our Int 2F start
        mov     ah,35h                  ;been modified
        mov     al,2fh
        int     21h                     ;call dos
        mov     ax,bx
        cmp     ax,cx
        jne     remove_error
```

If the two addresses are not exactly the same, then we know that another TSR program has grabbed this interrupt vector for its own purposes. In this case, SAFE is unable to remove the program from memory and therefore branches to the label REMOVE_ERROR. The two lines at this label simply set the Carry Flag and return to the statement after CALL REMOVE earlier in the program listing.

```
remove_error:
        stc                             ;show we had an error
        ret                             ;return to caller
remove  endp
```

On the other hand, if the two addresses are identical, then it is assumed to be OK to uninstall SAFE from memory at this time.

The first step that must be done to remove the program is to use Function 49h of Int 21h. This DOS function releases a block of memory specified in ES back to the system pool.

```
        pop     es
        mov     ah,49h
        int     21h                     ;call dos
        jc      remove_error            ;go if error occurs
```

Again, if the function is not able to release the block of memory, the Carry Flag is set by the DOS call and the program branches to the error routine.

If the function was successful, the original address of the procedure that processes Int 2Fh calls is restored. If you will recall, when SAFE is first installed, it saves the original vector address in the doubleword variable OLD2FH.

```
        push    ds                      ;save DS for now
        mov     ax,word ptr [old2fh+2]
        mov     ds,ax                   ;reset the segment of old
        mov     ah,25h                  ;set Int 2fh to
        mov     al,2fh                  ;original code
        mov     dx,word ptr [old2fh]
        int     21h                     ;call dos
        pop     ds                      ;restore DS now
        assume ds:codesg
        not word ptr es:[start]         ;get rid of ascii data
        clc                             ;show no errors
        ret                             ;return to caller
```

Function 25h of Int 21h, Set Interrupt Vector, is used to reset the Int 2Fh vector back to its original routine. This function call expects three parameters: the function code (25h) is stored in AH, the interrupt number of the interrupt to be set is loaded in AL, and DS:DX holds the segment and offset address of the interrupt service routine. In this case, the original segment and offset was stored in the OLD2Fh variable previously.

After DOS executes this call, the next step for SAFE is to erase the data stored at the label START. This data, which is two bytes in length, is required by all TSR programs that modify the Multiplex Interrupt. Microsoft calls this word of data a multiplex identifier. The identifer, which must be in the range C0h through FFh, is simply a function code.

This function code, placed in AH, is checked by each multiplex handler in the Int 2Fh chain, and the handler processes this service request accordingly. In other words, what we're doing here is creating our own DOS function call.

Identifying Interrupt Handler Requests

In order to understand this a bit better, let's skip back to part of SAFE's installation routine, as shown below:

```
check_6: mov    dx,offset copyw     ;point to copyright message
         mov    ah,09h              ;display string function
         int    21h                 ;call dos
         mov    ax,5453h            ;TSR check value
         int    02fh                ;go call current user
         cmp    al,53h              ;installed already?
         jne    tsr_error           ;no, error!
```

The purpose of this code fragment is to find out if SAFE is already installed in memory. This is done by issuing an Int 2Fh function call, with AX set to 5453h. The "5453h" value is the multiplex identifer we assigned to SAFE's interrupt handler HEADER. The value in AL is checked against the possible codes (in this case, 53h and 55h) to determine what action should be undertaken next.

At the label HEADER, which gets invoked every time an Int 2Fh occurs, SAFE checks the AX register. If AX is equal to 5453h, then it's a request to install SAFE in memory. If AX is equal to 5455h, then it's a request to remove the program from memory. In the event that AX does not equal either of these choices, control carries on at the label OK_STAY as usual to process the AEh function calls.

Summary

SAFE is a good example of using undocumented function calls to manipulate how the operating system works. By examining the program listing, you have learned to:

- change and modify interrupt service routines
- remove a TSR program from memory
- use two undocumented DOS function calls

- determine if a TSR has already been installed in memory previously
- retrieve switches or parameters from the command line when a program is executed
- determine the actual version of DOS installed in the computer system

Projects

1. Modify SAFE to disable a different DOS command, such as DELETE or COPY.
2. Modify SAFE to add an entirely new command to DOS. For example, you could add a command to send a specific initialization string to the printer.
3. Modify SAFE to disable more than one command at the same time. The user could specify the commands to be disabled on the command line.

For Further Study

If you want to learn more about using undocumented DOS function calls, consult Addison Wesley's book *Undocumented DOS.*

In addition, *PC Magazine* published an article and small utility that demonstrates how to add new commands to DOS using the two undocumented Int 2Fh function calls used in SAFE. This article appeared in Volume 10, Number 22, dated December 31, 1991.

Another article, also published in *PC Magazine*, explains the use of other undocumented DOS function calls (Volume 10, Number 3, dated February 12, 1991).

Programmer's Journal (July/August 1988, Volume 6.4) published a very clear explanation for using the Multiplex Interrupt 2Fh.

Program Listing

```
;SAFE.ASM
;<c> 1992 by Deborah L. Cooper
;
;This utility, available for DOS 3.3 or higher, disables the MSDOS
;FORMAT command. To allow someone to actually perform the FORMAT command,
;type the command: SAFE /U.
;
codesg  segment para public 'code'
        assume  cs:codesg
        org     100h                  ;a standard origin for com files
start:  jmp     init                  ;go make SAFE resident in memory

old_2f: db     0eah                   ;this byte represents a far jump
old2fh  dd      ?                     ;old Int 2Fh routine
[needs to be                              ;here!!!]
        dw      0                     ;just a cushion
command:db     'FORMAT',0             ;this is the command to disable
request db      0                     ;the terminator switch
msg:    db     0dh,0ah,07h,'I can't let you do that!',0dh,0ah,024h,00h
safe    equ    $+1000                 ;this is our memory protect size
        nop
;
;==============================================================
;Every time an Int 2Fh is issued by DOS, control comes through here.
;If the function is not AE00h or AE01h, then a jump is made to the
;original Int 2Fh handler
;==============================================================
;
stat_call:
        mov     al,0                  ;tell our status is "HERE!"
        iret                          ;and give to the checker
header: cmp     ax,5453h              ;are we checking for ourselves?
        jz      stat_call             ;no, then it's COMMAND.COM calling
        cmp     ax,5455h              ;is it request to leave?
        jne     ok_stay               ;if not carry on
        mov     request,1             ;say we want to leave now
        jmp     reply                 ;else quit
ok_stay:cmp     ah,0aeh               ;look for our calling
        jnz     old_2f                ;not us, let COMMAND.COM finish
        cmp     al,01h                ;is it our calling?
        jz      reply                 ;yes, do our routine then
        pushf                         ;save their flags
;
;==============================================================
;Register SI points to the parsed command that is currently being
;processed by COMMAND.COM in the format: byte count / text of command
;in uppercase
;==============================================================
;
isus:   push    si                    ;save their stuff
        push    cx                    ;for later
        mov     ch,0                  ;get command size in CX
        mov     cl,byte ptr [si]       ;from the buffer
        inc     si                     ;move up text of command
        push    bx                     ;save their stuff
        mov     bx,offset cs:command   ;point to our command
check:  mov     al,byte ptr [si]       ;are the two characters the
        inc     si                     ;same in both buffers?
        cmp     al,cs:[bx]             ;check ours
        jnz     no_com                 ;if not, let COMMAND.COM process
                                       ;this command
        inc     bx                     ;move up to next byte to check
        loop    check                  ;check the whole word
        mov     ax,0aeffh              ;report our status "it's ours!"
        jmp     go_com                 ;tell COMMAND.COM we are here
reply:  pushf                          ;save the flags
        mov     es:word ptr [si],0     ;set count byte to zero to tell
                                       ;COMMAND.COM we will process
                                       ;this command ourselves
        cmp     request,1              ;are we to terminate?
        jnz     do_format              ;if not, go do the other thing
        popf                           ;recover the flags (not needed)
        call    remove                 ;uninstall SAFE now
        mov     dx,offset msg2         ;show uninstalled message
        jc      er_exit                ;can't do it for some reason
        mov     dx,offset msg3         ;was un-installed ok message
        mov     ah,09h                 ;display string function
        int     21h                    ;call dos
        iret                           ;all done and gone!
er_exit:mov     ah,09h                 ;display string function
        int     21h                    ;call dos
        mov     ah,4ch                 ;terminate program function
        int     21h                    ;call dos
;
;==============================================================
;Instead of letting DOS process the 'FORMAT' command, we want to
;display the message 'I can't let you do that!' instead. This
;routine, in effect, replaces the 'FORMAT' routine
;==============================================================
;
do_format:
      mov     bx,offset cs:msg         ;get our response
      mov     ah,02h                   ;display byte function
show: mov     dl,cs:byte ptr [bx]      ;get one byte now
      cmp     dl,024h                  ;have we reached end of MSG?
      je      done                     ;yes, then exit
      int     21h                      ;no, display this byte
      inc     bx                       ;bump buffer pointer
      jmp     show                     ;do the next
done: mov     al,00h                   ;report our status
      popf                             ;reset the flags
      iret                             ;back to command.com
;==============================================================
;The command entered on the command line was not "FORMAT.";
;Therefore, let DOS execute it in the usual fashion
;==============================================================
;
no_com:
      mov      ax,0ae00h                  ;say "it's not for us!"
go_com:
      pop      bx                         ;restore all the
      pop      cx                         ;registers as they were before
      pop      si                         ;we interrupted this call
      popf                                ;get their flags back too
      jmp      old_2f                     ;and do original handler
nope: mov      al,00h                     ;tell them it's good
      jmp      old_2f                     ;go to old routine
;
;==============================================================
;REMOVE attempts to remove the resident copy of SAFE from
;memory, if possible
;==============================================================
;
remove  proc    near
        push    es
        mov     cx,offset header          ;point at our INT 2F
start
        mov     ah,35h                    ;been modified
        mov     al,2fh
        int     21h                       ;call dos
        mov     ax,bx
        cmp     ax,cx
        jne     remove_error
        pop     es
        mov     ah,49h                    ;of original program
        int     21h                       ;call dos
        jc      remove_error              ;go if error occurred
        push    ds                        ;save DS for now
        mov     ax,word ptr [old2fh+2]
        mov     ds,ax                     ;reset the segment of old
        mov     ah,25h                    ;set Int 2Fh to
        mov     al,2fh                    ;original code
        mov     dx,word ptr [old2fh]
        int     21h                       ;call dos
        pop     ds                        ;restore DS now
        assume  ds:codesg
        not     word ptr es:[start]       ;get rid of ascii data
        clc                               ;show no errors
        ret                               ;return to caller
remove_error:
        stc                               ;show we had an error
        ret                               ;return to caller
remove  endp

last:   dw      0                         ;marks end of resident code
lastbyte        = $                       ;as being here

copyw   db      'SAFE Utility',0dh,0ah
        db      '<c> 1992 by Deborah L. Cooper',0dh,0ah,0dh,0ah
        db      'This utility prevents a user from using the
```

```
MSDOS',0dh,0ah
        db      'command: FORMAT. SAFE requires MSDOS 3.3 or higher'
        db      0dh,0ah,0dh,0ah,'$'

errmsg  db      'SAFE requires DOS 3.3 or higher to function',0dh,0ah,'$'
msg2    db      'Cannot un-install SAFE','$'
msg3    db      'SAFE un-installed from memory now','$'
msg4    db      'SAFE already installed','$'
memmsg  db      'Unknown error encountered','$'

;=============================================================
;INIT is responsible for making SAFE resident in memory. It
;also checks to make the computer system is running DOS 3.3 or
;higher, since the AE00h and AE01h function calls are only
;available since DOS 3.3
;=============================================================
;
init    proc    near
        assume  cs:codesg,ds:codesg
;=============================================================
;Check DOS version. Must be 3.3 or higher
;=============================================================
dos:    mov     ah,30h                  ;get DOS version function
        mov     al,00h                  ;to check
        int     21h                     ;call dos
        cmp     al,3                    ;is it 3 or higher?
        ja      check_2                 ;go if 4 or greater
        jb      no_dos                  ;if less than 3, no go!
        cmp     ah,01Eh                 ;is minor version .30 ?
        jae     check_2                 ;no, can't do it then
no_dos: mov     dx,offset errmsg        ;point to error message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
;=============================================================
;See if a copy of SAFE is already installed in memory
;=============================================================
check_2:mov     si,81h                  ;point SI to command line
check_4:lodsb                           ;get the byte there
        cmp     al,20h                  ;we skip spaces!
        je      check_4                 ;now
        cmp     al,0dh                  ;exit if this is the end
        je      check_6                 ;of command tail
        cmp     al,'/'                  ;is it slashbar?
        jne     check_6                 ;if not, go install SAFE
        lodsb                           ;get next byte in AL
        and     al,0dfh                 ;convert to uppercase
        cmp     al,'U'                  ;want to un-install SAFE?
        jne     check_6                 ;no, go install SAFE in memory
uninstall:
        mov     ax,5453h                ;setup TSR check code
        int     02fh                    ;go call our TSR
        cmp     al,0                    ;installed already?
        jne     i_error                 ;no, error!
        mov     ax,5455h                ;un-install SAFE request
        int     02fh                    ;go un-install it
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
i_error: mov    dx,offset memmsg
error_exit:
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ah,4ch                  ;terminate program function
        int     21h                     ;call dos
tsr_error: mov  dx,offset msg4
        jmp     error_exit
;==============================================================
;Display copyright notice and make SAFE resident in memory now
;==============================================================
check_6:mov     dx,offset copyw         ;point to copywrite message
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
        mov     ax,5453h                ;TSR check value
        int     02fh                    ;go call current user
        cmp     al,53h                  ;installed already?
        jne     tsr_error               ;no, error!
        mov     ah,35h                  ;get interrupt 2Fh vector
        mov     al,2fh                  ;2fh code
        int     21h                     ;call dos
        mov     word ptr [cs:old2fh],bx ;save offset and segment
        mov     word ptr [cs:old2fh+2],es
        mov     ah,25h                  ;point to new routine
        lea     dx,header               ;to install
        int     21h                     ;call dos
init_done:
         mov     dx,(offset lastbyte - offset codesg + 15) shr 4
         mov     ax,3100h
         int     21h
         nop
init    endp

codesg ends                             ;end of code segment
        end     start                   ;end of program
```

Chapter 8

CAPSLOCK

From the DOS command line, the CAPSLOCK utility produces normal characters, regardless of whether the CapsLock key is engaged or not.

Like the previously discussed program TRAPBOOT, the utility developed in this chapter also monitors the computer system's keyboard activity. Many TSR programs are "turned on" by keyboard activity, and it's important to understand that this can be accomplished in different ways. The difference between TRAPBOOT and CAPSLOCK, however, is that CAPSLOCK must determine the status of the shift keys by calling Function 02h of Int 16h, Get Keyboard Flags. The TRAPBOOT program monitored the actual keystrokes typed on the keyboard, but CAPSLOCK also shows that any keystroke received, such as the letter "a," can be changed very easily by a TSR program, and the operating system doesn't even know or care that the character was modified.

Manipulating the Keyboard

The IBM PC's keyboard is a fascinating device that can be manipulated in a variety of ways. Consider, for example, the problem connected with the CapsLock key.

When the CapsLock key is engaged (toggled on), all keystrokes typed on the keyboard are automatically converted to uppercase. However, a problem occurs when you inadvertently toggle the CapsLock key on without knowing it. You then end up with a bunch of reversed uppercase and lowercase letters—exactly the opposite of your expectations!

The utility presented here is called CAPSLOCK. CAPSLOCK is a TSR utility that reprograms the keyboard so you can produce the desired results even though the CapsLock key is toggled on. In short, when the

CapsLock key is engaged, the correct keypress will be returned as if the CapsLock key was not really locked down.

Functions Used in CAPSLOCK.ASM

Int 16h, AH=02h	Get keyboard flags
Int 21h, AH=25h	Set interrupt vector
Int 21h, AH=31h	Terminate and stay resident
Int 21h, AH=35h	Get interrupt vector

How to Use CAPSLOCK

To run CAPSLOCK, you simply type the program's name at the DOS prompt or put it in your AUTOEXEC.BAT file so it is installed in memory whenever you boot your computer system. CAPSLOCK needs less than 200 bytes of memory and will not interfere with other TSR programs.

Tinkering with Keystrokes

When a key has been pressed on the keyboard, the BIOS interrupt 09h handler reads the hardware Port 60h for the keyboard and stores the scan and ASCII codes for the key in the keyboard buffer. This same interrupt handler modifies the Shift Key Status byte located in the BIOS Data Area if it detects that one of the shift or toggle keys has been pressed or released. Therefore, by intercepting Int 09h, CAPSLOCK can monitor each keypress before it is passed on to Int 09h for processing.

You can find a complete description of the BIOS Data Area and how the keyboard does the work of processing key presses and releases in Chapter 4's TRAPBOOT program.

Because the status of the keyboard is maintained by the BIOS whenever it detects a change, a program can determine if a certain toggle or shift key has been pressed simply by reading the Shift Key Status byte in the BIOS Data Area.

The byte at address 0040:0017h of the BIOS Data Area determines the setting of the CapsLock key. When bit 6 is set to a value of 1, the CapsLock key can be toggled on; similarly, when bit 6 is set to a value of 0, the CapsLock key can be toggled off.

From the table below, you can see that a number of other shift keys such as Insert can also be manipulated in this fashion. In addition, you can also read the status of any of these shift keys by using Function 02h of Int 16h, Get Keyboard Flags. To do this, the AH register is loaded with the function code (02h), and the Int 16h instruction is executed. On return from this function, the AL register will contain a value that describes the state of each shift key. This is exactly the same information depicted in the table below.

Right shift down	0000 0001
Left shift down	0000 0010
Ctrl down	0000 0100
Alt down	0000 1000
ScrollLock depressed	0001 0000
NumLock depressed	0010 0000
CapsLock depressed	0100 0000
Insert on	1000 0000

CAPSLOCK uses Function 00h of Int 16h to retrieve the keyboard status byte, and it uses direct access techniques to save the modified status byte back in the BIOS Data Area. This is done because there is no DOS or BIOS function call that can manipulate the state of the shift or toggle keys directly.

Redirecting the Keyboard Vector

The INITIALIZE routine shown below is responsible for making CAPSLOCK resident in memory and for redirecting the keyboard vector.

```
initialize proc near
        assume  cs:codesg,ds:codesg       ;set up the
        mov     bx,cs                     ;code and data
        mov     ds,bx                     ;segments first
        cli                               ;do not allow interrupts just now
        mov     al,09h                    ;number for keyboard
        mov     ah,35h                    ;get interrupt vector function
        int     21h                       ;call dos
        mov     old_kybrd,bx              ;save the offset address
        mov     old_kybrd[2],es           ;save the segment address
        mov     dx,offset start           ;point to our new routine
        mov     ah,25h                    ;set interrupt vector function
        mov     al,09h                    ;keyboard interrupt vector
        int     21h                  ;call dos

        mov     dx,(offset end_prog - offset codesg + 15) shr 4
        mov     ax,3100h             ;terminate and stay resident
        int     21h                  ;call dos
initialize endp
```

By now the code shown above should look very familiar to you. Almost every single TSR program you write will use Function 35h, Get Interrupt Vector, and Function 25h, Set Interrupt Vector. These two service calls insert a new interrupt routine into the PC's memory. In addition, all TSR programs must set aside a block of memory large enough to hold the TSR's code and data areas. Function 31h of Int 21h is then used to load the TSR into memory.

Building a Better Routine

At the label START, the extra segment (ES) register is set to the ROM BIOS Data Area. As discussed earlier, the ROM BIOS Data Area of the PC holds information relevant to the keyboard, as well as other system information. With regards to the CAPSLOCK utility, location 0040:0017h is of particular interest to us at this time. This byte in memory will let us know if the CapsLock key is toggled on or off.

By calling Function 02h of Int 16h, Get Keyboard Flags, our utility can determine the state of the CapsLock key. This is done with the following code:

```
        mov     ah,02h               ;get shift status
        int     16h                  ;call bios
caps:   test    al,01000000b         ;is CapsLock pressed?
        jnz     shift                ;yes, continue then
        jmp     exit                 ;no, just do normal keyboard
```

Once the Int 16h function has been executed, the AL register contains the status of all shift keys. Therefore, by using the TEST instruction we can determine if bit 6 is on or off. If bit 6 equals 0, then the CapsLock key is not engaged and the program branches to the EXIT routine. In this case, none of the characters typed on the keyboard are modified by our utility.

However, if bit 6 equals 1, then we know that the CapsLock key is toggled on and the program branches to the label SHIFT.

At the label SHIFT, we need to find out if one of the shift keys, left or right, is also depressed. If the user isn't trying to produce an uppercase character, then we let control pass on to the normal keyboard routine.

```
shift:  test    al,00000011b            ;is a shift key down?
        jz      exit                    ;no, just do normal keyboard routine
```

If the user pressed either the left shift or right shift key, then the next step CAPSLOCK does is to read the keyboard. Since we might want to manipulate the character typed on the keyboard before it reaches the BIOS, we need to read the scan code directly. This is done with the statement:

```
        in      al,60h                  ;get scan code from Port A (keyboard)
```

The "60h" in this statement refers to Port 60h. This port is simply an interface to the keyboard hardware and therefore is available for use by programmers.

To continue, the above instruction would read the next keystroke typed by the user and store the result in the AL register. We are then free to test the value in AL to see if it is in the range A through Z. If the character is in this group, CAPSLOCK jumps to the label EXIT and no change is made.

If the character is not in the A-Z range, then control jumps to the code starting at the label ADJUST.

```
adjust: and     ah,00000000b            ;set CapsLock off for this keycode
        mov     es:[417h],ah            ;save it
```

The AND instruction above is used to turn the CapsLock bit (bit 6) to zero. The next instruction (mov es:[417h],ah) stores this new value in the ROM BIOS Data Area, which in turn fools the PC into believing that the CapsLock key is not really engaged at all. The actual character the user typed on the keyboard is then passed on to the original keyboard routine for further processing.

Summary

CAPSLOCK is a very small but useful program. With it, you have learned:

- to change and modify interrupt routines for the keyboard
- to directly access and modify the status bytes for the special shift and toggle keys in the BIOS Data Area

Projects

1. Modify CAPSLOCK to accept a command line parameter (such as "/U") to uninstall the program from memory. Use one of the other TSR utilities presented in this book as a model when writing the routine to remove the program from memory.
2. CAPSLOCK could be used to toggle the state of the CapsLock, NumLock and ScrollLock keys by allowing the user to specify choices on the command line. For example, CAPSLOCK /C+ /N- /S+ would tell your program to toggle CapsLock on, NumLock off, and ScrollLock on.

For Further Study

PC Magazine published a two-part series explaining how the keyboard works and how to use many of the DOS and BIOS functions to manipulate the keyboard (Volume 9, Number 22, dated December 25, 1990; Volume 10, Number 1, dated January 15, 1991).

Program Listing

```
;CAPSLOCK.ASM
;<c> 1992 by Deborah L. Cooper
;
;Memory resident utility to make the keyboard perform as usual
;even when the CapsLock key is depressed.
;
codesg  segment                         ;start of program
        assume cs:codesg
        org     100h                    ;make this a COM file
begin:  jmp     initialize              ;make it memory resident first

old_kybrd dw 2 dup(?)                   ;original keyboard interrupt routine
qkey    equ     10h                     ;scan code for Q key
pkey    equ     19h                     ;scan code for P key
akey    equ     leh                     ;scan code for A key
lkey    equ     26h                     ;scan code for L key
zkey    equ     2ch                     ;scan code for Z key
mkey    equ     32h                     ;scan code for M key

start   proc    near
        sti                             ;enable interrupts
        push    ax                      ;we must first save
        push    cx                      ;all the registers
        push    bx                      ;which will be restored
        push    dx                      ;when we exit back to
        push    di                      ;the normal keyboard
        push    es                      ;routine
        pushf                           ;save the flag register

        mov     ax,0                    ;set up the ES segment
        mov     es,ax                   ;to this one              Not
        mov     ah,02h                  ;get shift status         needed
        int     16h                     ;call bios
;
;===========================================================
;Determine if the CapsLock key is engaged
;===========================================================
;
caps:   test    al,01000000b            ;is CapsLock pressed?
        jnz     shift                   ;yes, continue then
        jmp     exit                    ;no, just do normal keyboard
                                            ;routine
;
;===========================================================
;Determine if the left shift or right shift key is depressed
;===========================================================
;
shift:  test    al,00000011b            ;is a shift key down?
        jz      exit                    ;no, just do normal keyboard routine
        in      al,60h                  ;get scan code from Port A (keyboard) in
;Is the key code in  the range P - Q?                                       a l
        cmp     al,qkey                 ;is the keycode less than Q key?
        jb      exit                    ;yes, ignore it then
        cmp     al,pkey                 ;is the keycode less/equal to P key?
        jna     adjust                  ;yes, adjust as its P-Q keys
;Is the key code in the range A - L?
        cmp     al,akey                 ;is the keycode less than A key?
        jb      exit                    ;yes, then ignore it
        cmp     al,lkey                 ;is the keycode less/equal to L key?
        jna     adjust                  ;yes, adjust as its A - L keys
;Is the key code in the range M - Z?
        cmp     al,zkey                 ;is the keycode less than Z key?
        jb      exit                    ;yes, ignore it then
        cmp     al,mkey                 ;is the keycode less/equal to M key?
        ja      exit                    ;yes, adjust as its M - Z keys
adjust: and     ah,00000000b            ;set CapsLock off for this keycode
        mov     es:[417h],ah            ;save it
exit:   popf                            ;restore registers
        pop     es                      ;now restore all the
        pop     di                      ;used registers
        pop     dx                      ;exactly as they
        pop     bx                      ;were before we
        pop     cx                      ;called on our
        pop     ax                      ;new routine
        jmp     dword ptr cs:[old_kybrd]   ;do original keyboard routine
start   endp

end_prog = $
```

codesg

```
;
;================================================================
;INITIALIZE is the routine that puts CAPSLOCK into memory as TSR
;================================================================
;
initialize proc near
        assume  cs:codesg,ds:codesg   ;set up the
        mov     bx,cs                 ;code and data
        mov     ds,bx                 ;segments first
        cli                           ;do not allow interrupts just now
        mov     al,09h                ;number for keyboard
        mov     ah,35h                ;get interrupt vector function
        int     21h                   ;call dos
        mov     old_kybrd,bx          ;save the offset address
        mov     old_kybrd[2],es       ;save the segment address
        mov     dx,offset start       ;point to our new routine
        mov     ah,25h                ;set interrupt vector function
        mov     al,09h                ;keyboard interrupt vector
        int     21h                   ;call dos

        mov     dx,(offset end_prog - offset codesg + 15) shr 4
        mov     ax,3100h              ;terminate and stay resident
        int     21h                   ;call dos
initialize endp

codesg  ends                          ;end of code segment
        end     begin                 ;end of our program
```

Chapter 9

ICU

ICU displays a highlight bar across the screen where the cursor is located. This program is useful on laptop and notebook computers, as well as standard desktop machines, where you cannot easily see the cursor.

ICU is the final program to be presented in this book. Again, this TSR program will teach you something brand new. ICU is the only program in this book that writes directly to the video screen instead of through the BIOS functions. This is an important concept to master since direct video access can produce the fastest output possible while also providing direct control over a screen's appearance. To this end, ICU will teach you how to determine the type of video card installed in a computer system, how to determine when it is safe to interrupt DOS to perform your own work, and how to create special effects such as highlight bars.

ICU is also more advanced than the other utilities presented in this book simply because it intercepts the video (Int 10h) and the keyboard (Int 09h) functions at the same time. In our other programs, only one—not both—were manipulated by our own TSR routines.

Where Are You?

If you've ever used a laptop or notebook computer, then you will appreciate ICU from both a user's point of view and from a programmer's point of view. ICU, when instructed to do so, displays a highlight bar across the screen where the cursor is located. It makes the cursor very visible and easy to locate.

ICU is certainly the most complicated utility presented in this book simply because it monitors both the keyboard and the video interrupt services (09h and 10h, respectively). In addition, this TSR utility can be temporarily toggled off and then back on again. The other TSR utilities in this book must be removed from memory—they cannot be switched on or off at will.

Functions Used in ICU.ASM

Int 10h, AH=0Eh	Display byte
Int 16h, AH=02h	Get shift key status
Int 21h, AH=09h	Display string
Int 21h, AH=25h	Set interrupt vector
Int 21h, AX=3100h	Terminate and stay resident
Int 21h, AH=34h	Get address of INDOS flag
Int 21h, AH=35h	Get interrupt vector
Int 21h, AH=49h	Release block of memory
Int 21h, AH=4Ch	Terminate process with return code

How to Use ICU

To run ICU, you simply type its name at the DOS prompt or put it in your AUTOEXEC.BAT file so it is installed in memory whenever you boot your computer system. ICU needs less than 1K of RAM and should not interfere with other TSR programs.

Once ICU has been installed in memory, it can be activated by pressing the Alt and period (.) keys together. A highlight bar will immediately be displayed at the cursor's current row and column position. If you use the up or down arrow keys to move the cursor, the highlight bar will also move. ICU works with any application program or at the DOS prompt.

You may toggle the highlight bar on and off by pressing the hotkey Alt+period combination a second time. However, to remove ICU from memory you must use the command ICU /U to completely remove it from memory.

A Little Housekeeping

Several other memory resident utilities presented earlier in this book show how a TSR is made resident in memory. In addition, the routines used to remove the TSR from memory are virtually the same as those used by ICU. After all, why should you spend a great deal of time reinventing and debugging code that already works in a different program? The only difference between ICU and the other programs in this book is that code must be added to handle both the Int 09h and Int 10h routines.

If you look closely at the code starting at the label NOT_IN shown below, you'll see a neat trick. At this point in the initialization routine, we are trying to determine if the command line contains any parameters.

```
not_in: mov     si,80h                  ;address of command line
        lodsb                           ;get length in AL
        or      al,al                   ;are there any switches?
        jz      make_r                  ;no, go install then
```

Once the AL register contains the byte read from address 80h within the PSP (the length of the command line), we need to test its value. Notice the OR AL,AL statement used here. When a value is ORed with itself, each bit in the first operand is set to 1 if the corresponding bit in either or both of the operands equals 1. Therefore, the Zero Flag (ZF) will be set to 1 if all the bits in AL are 0. If all the bits are not 0, then the ZF is set to 0. This is an excellent technique you can use to see if a value is equal to zero.

What Kind of Video Card is Installed?

ICU displays its horizontal bar across the screen so quickly that it appears to be "just there." The technique of writing directly to video memory is used in many types of programs, including TSRs. We will now discuss how a program like ICU can manipulate the screen to produce results instantaneously.

The memory of a PC, as you know, is divided into various blocks. Each of these blocks is used by the operating system to hold information about the physical aspects of the hardware, the interrupt vector table, and the PC's video refresh buffer.

The video refresh buffer, usually located on a display adapter board, is an area of memory set aside to hold the contents of the screen. To write a character to the screen, you only need to store that ASCII character to a specific location within the video buffer and it will instantly appear on the display.

The segment address of the video refresh buffer depends on a number of factors. First of all, there are two video modes available on the PC—text mode and graphics mode. In this discussion, we will concern ourselves only with text mode operation.

Once the mode, in this case its text only, has been determined, there are two segment addresses that can be used. For text mode on a monochrome system, the segment address is B000h, and on a color system, the address is B800h. Therefore, the first step is to determine what video mode is currently being used on the PC.

Since ICU manipulates the screen's contents in order to display the horizontal bar where the cursor is located, we must determine what kind of video adapter is installed in the computer system. The BIOS Data Area holds this information.

```
make_r: mov     ax,40h                  ;AX = BIOS segment
        mov     es,ax                   ;into ES
        mov     al,es:[49h]             ;get video mode
        cmp     al,7                    ;is it mode 7?
        jne     init_3                  ;no, then its a color system
init_2: sub     video_segment,800h      ;set for monochrome
init_3: mov     bar_status,1            ;toggle bar to ON to start with
        mov     ax, es:[50h]            ;get cursor address from BIOS
        mov     orig_row,ah             ;save the ROW value for later
```

In the code fragment above, the current video mode is retrieved from the BIOS Data Area into the AL register. If the value in AL is seven, then we know that this is a monochrome system and the program continues executing at the label INIT_3. However, if the value in AL is not seven, then we assume that a color system is installed and we set the video segment address to B800h.

When it comes time for ICU to display or erase the highlight bar, the actual offset address within the segment address of the video buffer is calculated. This is the exact memory location used by ICU to store the new screen attribute.

In text mode there are normally 2,000 memory locations within the video memory buffer. Each position on the physical screen is comprised of one byte for the ASCII character and one byte for the character's attribute. There are 80 positions on one screen row; this translates to 160 bytes per screen row. This technique of writing directly to video memory will be discussed in more detail later in this chapter.

The DOS Reentrancy Problem

MS-DOS is a non-reentrant operating system. This means that only one task can be performed on the computer system at one time. Since we can write TSR programs that are activated by various means, we do know that a trick is available to have two or more programs operating at the same time.

When writing memory-resident software for the PC, you cannot use any DOS service calls, such as Int 21h functions, because DOS is non-reentrant. You are, however, free to use any and all of the BIOS function calls. Using BIOS services within a TSR program does not cause any conflicts. The problems only occur when using DOS function calls.

Now that you've just been told you can't use DOS functions, you might be wondering how some TSRs open files, check the keyboard for input, and so on. The trick to doing these kinds of DOS functions is to use some undocumented, but well known, DOS functions. This does add quite a bit of overhead programming, but your TSR can perform any function you desire.

An undocumented DOS function can be used to determine if it is safe to perform a DOS service call. This special function, which is extremely important when writing memory resident utilities, is called Get INDOS Flag.

When the Get INDOS Flag, Function 34h of Int 21h, is called, it returns a pointer in ES:BX to a byte in memory called the MS-DOS Busy Flag. If this byte's value is not equal to zero, then the computer system is in the process of executing an MS-DOS function and it cannot be interrupted. However, if this byte is equal to zero, then its assumed that DOS is not busy and it is safe to perform a DOS service call.

Before you can use the Get INDOS Flag service within a TSR program, the initialization routine must retrieve and save the address of the INDOS Flag. This is done with the following code fragment:

```
mov     ah,34h                  ;get address of
int     21h                     ;the INDOS flag
mov     word ptr indos,bx       ;save offset address
mov     word ptr indos[2],es    ;save segment address
```

Once the address of the INDOS Flag has been saved, we can check this byte's value when our TSR is called into action. Again, if the value of the INDOS Flag is not zero, we cannot perform any of our TSR's routines, except to return control to the system, as shown here:

```
cur_1:  push    di                      ;save DI
        push    es                      ;and ES
        les     di,cs:[indos]           ;exit if the INDOS
        cmp byte ptr es:[di],0          ;flag is non-zero
        jne     no_do                   ;exit NOW!
        pop     es                      ;restore ES
        pop     di                      ;and DI
        jmp     short cur_2             ;and do our routine
no_do:  pop     es                      ;restore ES
        pop     di                      ;and DI
        jmp     short exit              ;and quit altogether
```

This routine, starting at the label CUR_1, is done at the very beginning of our memory resident program. If we had done this later on in the program, we may have disrupted one or more of the registers currently being used by DOS if it is in the middle of executing a command. By checking the current status of the INDOS Flag immediately upon entering our TSR, we can be sure that we will not change any part of the system. In TSR programming, this is a crucial lesson to heed.

Chaining Interrupt Routines

ICU's own procedure for processing Int 10h and Int 09h calls must be inserted into the table of interrupt vectors. The following code fragment does this:

```
        mov     ah,35h                  ;get interrupt 10h vector
        mov     al,10h                  ;for BIOS and timer
        int     21h                     ;call dos
        mov     word ptr old10h,bx      ;save offset address
        mov     word ptr old10h[2],es   ;save segment address

        mov     ah,25h                  ;set interrupt vector function
        mov     dx,offset video         ;to this routine
        int     21h                     ;call dos

        mov     ah,35h                  ;get interrupt vector function
        mov     al,09h                  ;for the keyboard
        int     21h                     ;call dos
        mov     word ptr old9h,bx       ;save offset address
        mov     word ptr old9h[2],es    ;save segment address

        mov     ah,25h                  ;set interrupt vector function
        mov     dx,offset keybd         ;to this routine
        int     21h                     ;call dos
```

These lines of code perform a necessary procedure that is standard when writing TSR programs. The address of the old routines that process Int 10h and Int 09h service calls are first saved. This is done so that the routine that removes ICU from memory can restore these interrupts as they were before ICU was initially installed in the computer system. Function 35h of Int 21h, Get Interrupt Vector, is used to retrieve the interrupt vector's segment and offset address. This is returned in ES:BX, and we save these in the variables OLD10h and OLD9h for later processing.

Function 25h of Int 21h is then used to set the Int 10h and Int 09h services to ICU's own procedures. The DS:DX register pair is loaded with the VIDEO routine (for Int 10h) and the KEYBD routine (for Int 09h), and, once the Int 21h instruction is executed, our routines are placed in memory. From this point on, all calls by the system to Int 10h will first be processed by our VIDEO routine and all calls to Int 09h will be processed by our KEYBD routine.

Making it Resident in Memory

ICU uses Function 31h to reserve the memory it needs. This is accomplished with the following three lines of source code:

```
mov     dx,(offset end_prog-offset codesg + 15) shr 4
mov     ax,3100h                ;terminate function
int     21h                     ;call dos
```

As with several other programs in this book, ICU uses Function 31h to reserve enough room for itself in memory and then loads a copy of itself into that memory space. For more information on using Function 31h, please refer to the discussion about TRAPBOOT elsewhere in this book.

We have also taken the liberty of deallocating ICU's environment block. As has been previously discussed, the environment block is not really necessary to ICU's performance. Therefore, we should let the computer system use the 256 bytes of memory for other purposes.

Removing ICU from Memory

If the uninstall option was found on the command line, then the program branches to the label UNINSTALL.

```
uninstall:
        assume  ds:codesg
        push    es                      ;save ES first
        mov     ah,35h                  ;and see if ICU still
        mov     al,10h                  ;BIOS/timer interrupt
        int     21h                     ;owns it
        mov     ax,es                   ;compare vector address
        cmp     ax,res_segment          ;with ICU's code segment
        jne     un_1                    ;go if cannot uninstall
```

Next, ICU must make sure that the Int 10h vector has not been modified by another program since ICU was installed. The only way to determine this is to use Function 35h of Int 21h to retrieve the segment and offset address of the current service routine. The information will be returned in ES:BX. All ICU has to do is load the segment address of Int 10h's service routine into AX and compare it with ICU's segment, stored in the variable RES_SEGMENT.

If the segment address of the routine servicing Int 10h calls is the same as the segment address of ICU's code segment, then it is safe to assume that we can successfully remove the utility from memory. Otherwise, the program would branch to the label UN_1, and that code would display an error message and then exit to DOS with no other action being taken.

If ICU can be removed, the program continues by deallocating the block of memory used by ICU. Although an image of ICU will still remain in memory, it will be completely deactivated. The next program DOS loads will overwrite this area of the PC's memory.

```
        mov     ah,49h                  ;free memory function
        int     21h                     ;call dos
        jc      un_1                    ;go if error
restore:push    ds
        assume  ds:nothing
        lds     dx,es:[old10h]          ;DX=video interrupt address
        mov     ah,25h                  ;set interrupt vector address
        mov     al,10h                  ;for Int 10h
        int     21h                     ;call dos
        lds     dx,es:[old9h]           ;DX=keyboard interrupt address
        mov     ah,25h                  ;set interrupt vector address
        mov     al,09h                  ;for the keyboard
        int     21h                     ;call dos
        pop     ds
        assume  ds:codesg
        not word ptr es:[start]       ;remove fingerprint
        push    cs                    ;set DS to our
        pop     ds                    ;code segment again
        assume  ds:codesg
        mov     dx,offset out_now     ;ICU removed message
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        pop     es                    ;recover ES too
        mov     ah,4ch                ;terminate program function
        int     21h                   ;call dos
un_1:   mov     dx,offset no_way      ;cannot uninstall ICU!
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        pop     es                    ;restore ES now
        mov     ah,4ch                ;terminate program function
        int     21h                   ;call dos
init    endp
```

The first step we need to take to remove ICU from memory is to use Function 49h of Int 21h. This DOS function releases a block of memory specified in ES back to the system pool. If the function is not able to release this block of memory, the Carry Flag is set by the DOS call and the program branches to the label UN_1. The code at the label UN_1 simply displays an error message indicating that ICU cannot be removed from memory.

If the memory block was freed successfully, Function 25h of Int 21h is then used to set the video and keyboard vectors back to their original routines as they were before ICU was made resident in memory. The DS:DX register pair is loaded with the address of the VIDEO_INT and KEYBOARD_INT routines, and, once the Int 21h instruction is executed, the original routine is restored. Then the first two bytes of ICU's program is erased with the statement NOT WORD PTR ES:[START], and the STATUS flag is reset, to signal that ICU is no longer stored in memory.

Up and Running

When ICU is installed in memory and the PC detects that the hotkey combination has been pressed, the new routine for Int 09h is called into action. The procedure KEYBD gets executed every time the PC detects that a key has been pressed or released on the keyboard. In addition, whenever the system receives an Int 10h service request, the new procedure VIDEO is activated by ICU.

Precautions for TSRs

Many TSR programs that are activated by pressing a hotkey combination include a precaution against the TSR activating a second time. This can be prevented by using a special flag. When the TSR is initially activated, it sets a flag to say it is currently in operation. Then if a second hotkey combination is detected before the TSR has completed its work, the flag is reset to signal that it will be activated the next time the keystroke combination is received.

In ICU, the flag BAR_STATUS uses this technique to monitor itself. BAR_STATUS is initially set to a value of 1 when ICU is first installed in memory. When you press the hotkey combination Alt+period, ICU checks the value of BAR_STATUS. If BAR_STATUS is equal to 0, then the program erases the highlight bar from the screen. However, if BAR_STATUS is equal to 1, then the program displays the highlight bar at the cursor's current position on the screen. Therefore no matter how many times the hotkey is detected, BAR_STATUS will simply toggle itself on or off according to the BAR_STATUS flag setting itself.

Monitoring the Video Services

ICU monitors both the keyboard Int 09h and the video Int 10h services provided by the operating system. By trapping these two services, ICU is able to display its highlight bar at the cursor's current location on the screen instantaneously and without harming another application's work.

We'll discuss the VIDEO procedure first. This procedure itself is quite small and is shown here.

```
video   proc    near
        pushf                           ;needed to simulate Int
        call    cs:old10h               ;call for original routine
        cmp     cs:bar_status,1         ;do our CURSOR routine?
        jne     video_1                 ;no, skip it
        call    do_cursor               ;yes, do it now
video_1:iret                            ;all done, so exit
video   endp
```

The first step taken by this procedure is to store the flags register on the stack and then to call the original interrupt routine for Int 10h. In order to make this call to the original routine, it is absolutely necessary to push the flags onto the stack; the Int 10h service will automatically perform stack cleanup when the call is completed.

The instruction CALL VIDEO_INT tells the system to execute the original routine for Int 10h that was in place before ICU was installed in memory. In this way, all the BIOS video functions are allowed to operate in the usual manner to output characters to the display, update the cursor's position and so on without any intervention from ICU.

However, the next instruction compares the value of the variable BAR_STATUS to one. This variable is used as a flag in ICU; it signals whether the highlight bar should be toggled on (displayed) or toggled off (not displayed). Therefore, if BAR_STATUS is currently equal to 0, the program simply performs an IRET instruction and no further action is taken.

If, on the other hand, BAR_STATUS is equal to a value of 1, then ICU's DO_CURSOR routine is called. The DO_CURSOR routine displays the highlight bar. When this procedure is finished, the program jumps to the label VIDEO_1 to execute the IRET instruction. The BIOS isn't even made aware that another routine is also called every time the system executes an Int 10h service call!

Monitoring the Keyboard Services

The new routine for the keyboard, called KEYBD, works in almost the same way as the VIDEO routine described above.

ICU's KEYBD routine monitors every keystroke the system receives. To determine the status of the keyboard, the instruction IN AL,60h is used to retrieve the scan code of the key just pressed. If that key's scan code is 34h, corresponding to the period key, then ICU must make a further test for the Alt key.

If the key pressed is not the period key, ICU passes the scan code back to the original keyboard service routine, as if nothing had happened. Indeed, all TSR programs that intercept keystrokes in this manner must make a call to the original routine to enable other application programs and TSRs to process the keystroke.

On the other hand, if the scan code indicates the period key was pressed, then ICU's next task is to determine the state of the shift keys. This is done by using Function 02h of Int 16h, Get Keyboard Flags. On return from this service, the AL register holds the status of all toggle and shift keys. If bit 3 is set, then the Alt key is also pressed, which is ICU's hotkey combination.

```
keybd   proc    far
        sti                             ;turn interrupts on
        push    ax                      ;save AX
        in      al,60h                  ;get scan code from keyboard
        cmp     al,34h                  ;our period keystroke?
        jne     kb2                     ;no, do normal keyboard then
        mov     ah,02h                  ;check shift key status function
        int     16h                     ;call bios
        cmp     al,8d                   ;is it Alt key too?
        jne     kb2                     ;no, do normal keyboard then
        call    kb_reset                ;yes, ignore the keystroke
        cmp     cs:bar_status,1         ;is bar toggled on?
        je      yes_bar                 ;yes, go turn it off then
        mov     cs:bar_status,1         ;no, turn toggle switch to ON
        jmp     short kb1               ;and continue
Es     cs:bar_status,0                  ;turn toggle switch to OFF
kb1:    call    do_cursor               ;do our routine
        pop     ax                      ;restore AX
        iret                            ;end of interrupt routine
kb2:    pop     ax                      ;recover AX
        jmp     cs:old9h                ;do original keyboard routine
keybd   endp
```

If both the Alt and period keys were detected, then ICU resets the keyboard by calling the procedure KB_RESET. This tells the system to ignore the Alt+period keystroke combination. Next, ICU tests the variable BAR_STATUS.

BAR_STATUS, you will recall, is ICU's method of determining whether or not the highlight bar should be displayed or erased. In the KEYBD procedure, if this flag is set to a value of 1, the program branches to the label YES_BAR, which resets the BAR_STATUS flag and removes the highlight bar from the screen. If this flag is set to a value of 0, the program sets it to a value of 1. This signals the DO_CURSOR routine that it must display the highlight bar. In other words, whenever the Alt+period keystroke is detected, the value of BAR_STATUS is reversed from its current settings. This, in turn, toggles the highlight bar on or off accordingly.

Creating Special Effects

The procedure BAR is a very interesting one from a programming point of view. This routine determines the cursor's current row and column position on the screen. Next, it uses another procedure, WRCHAR, to change the attribute of each character located on this particular row where the cursor is located to another attribute.

ICU makes extensive use of a number of interesting programming techniques. First, the technique of directly accessing specific BIOS video parameters each time DO_CURSOR is executed ensures that ICU will work correctly each time it is called on to perform its work. This means that even if an application program changes the number of columns on the screen (video mode), ICU will work correctly. It will automatically adjust itself to any changed information in the BIOS Data Area.

Another interesting technique employed not only by ICU but also by many other TSR utilities is that of direct video access. In other words, ICU reads and writes information to and from the screen without calling any DOS or BIOS service calls. This enables ICU to produce the highlight bar very fast, far faster than could be achieved using the DOS or BIOS services. By bypassing DOS and BIOS service calls, the bar is displayed instantaneously.

Direct Access to Video RAM

ICU reads from and writes to the video RAM memory area of the PC. For text modes, the monochrome and Hercules video cards use video RAM from address B000:0000 to B000:7FFF. Color cards such as the EGA and VGA use video RAM starting from address B800:0000 and up, depending on how much memory the card supports. Any data written to this area of memory will immediately appear on the screen.

The data stored in video RAM consists of two bytes of information that make up one character position on the screen. The first byte, which is always an even address, is the ASCII code of the character you want to display. Appendix II shows a chart of the 256 characters that comprise the ASCII character set on the PC. All of these characters can be stored in video RAM.

The second byte of data you must also store in video RAM is the attribute byte. This byte determines how the ASCII character will be displayed. For example, each character could be set to a specific color on a color card system or to underline on a monochrome system. The attribute byte is always stored at an odd address and affects only one character. Therefore, if you want to change the look of an entire row of characters on the screen, all attributes for that specific row must be changed individually. You cannot just change one specific attribute.

Video Parameters in the BIOS

As has been mentioned elsewhere in this book, the BIOS Data Area holds information the BIOS accesses when communicating with programs. Although most of the information in this special area of memory is readily accessible through interrupt functions, ICU accesses these variables directly. As you can see from the code shown below, the DO_CURSOR procedure reads several variables from the BIOS Data Area.

The ES register is first set to the segment address of the BIOS Data Area. The DO_CURSOR procedure can then read several of the variables directly from the BIOS Data Area.

```
        mov     ax,40h                  ;point ES to the
        mov     es,ax                   ;Bios Data Area
        mov     al,es:[49h]             ;get video mode number in AL
        cmp     al,2                    ;is it mode 2?
        je      cur_1                   ;yes, proceed then
        cmp     al,3                    ;is it mode
        je      cur_1                   ;yes, proceed then
        cmp     al,7                    ;is it mode 7?
        je      cur_1                   ;yes, proceed then
```

This code fragment retrieves the current video mode from the BIOS Data Area, which is at address 0040:0049. The video mode is then tested to make sure it's one of the three possible text modes. ICU, along with most other TSR programs, will only work in text mode, not graphics mode. Therefore, if the video mode in AL is not equal to 2, 3, or 7, then ICU simply returns control to the calling function without doing its own work.

If the video mode is a valid one, the program branches to the label CUR_1. As was explained earlier, this section retrieves the address of the INDOS Flag for later use.

At the label CUR_2, which is shown below, ICU retrieves the current video page, the number of columns currently displayed, the number of bytes for one video page and the cursor's current address from the BIOS Data Area. All of these values are saved in separate variables and will be used later on in the program.

```
cur_2:  mov     al,es:[62h]             ;get current video page
        mov     video_page,al           ;save it
        mov     ax,es:[4ah]             ;get number of columns displayed
        mov     [num_cols],ax           ;for the video mode
        mov     ax,es:[4ch]             ;# of bytes for one video page
        mov     page_size,ax            ;(4,000 for 80 x 25 mode) etc
        mov     ax,es:[50h]             ;get cursor address from CRTC
        mov     cursor_pos,ax           ;save it
```

Once all the variables about the video screen have been retrieved from the BIOS Data Area and saved in ICU's own variables, ICU goes into action. The first thing that happens is that ICU checks the flag BAR_STATUS. If BAR_STATUS is equal to zero, then the currently displayed highlight bar must be turned off. In this case, the program branches to the label OFF_NOW, shown here.

```
off_now:xor     ah,ah                   ;set column to zero
        mov     al,orig_row             ;get row in AL now
        call    getpos                  ;calculate video address in DI
        mov     dl,old_attrib           ;set AL to original attribute
        call    bar                     ;turn the highlighted bar off!
        jmp     exit                    ;and exit
```

This section of code uses the XOR instruction to set AH to 0, which in effect tells ICU we want to start with column zero and the row location in ORIG_ROW. Therefore, on entry to the procedure GETPOS, which calculates the direct video address, AX holds the row and column coordinates of the highlight bar's current position on the screen.

The procedure GETPOS calculates the offset address in video memory that corresponds to the current row and column position of the cursor's location. On entry to this subroutine, AX holds the row and column position of the cursor and BX holds the video page number. When the procedure has been executed, the AX register will hold the direct video address that we will be writing data to.

```
getpos  proc    near
        mov     bx,ds:[num_cols]        ;we must multiply the row
        shl     bx,1                    ;value by # bytes per video row
        mul     bl                      ;multiply columns by 2
        xor     bh,bh                   ;convert DX to a word now
        mov     bl,video_page           ;add results
        push    cx
        mov     cx,ax                   ;row and column in CX
        mov     ax,page_size            ;length of a video page
        mul     bx                      ;multiply page length by # pages
        add     ax,cx                   ;AX=video address
        pop     cx
        ret                             ;return to caller
```

GETPOS is a neat little procedure. Most likely you will use this in a number of utilities when you need to read or write data in the video memory of the PC. The first thing the procedure does is to retrieve the number of columns for this particular video mode into BX. Then the instruction SHL BX,1 multiplies this value by the number of bytes per row on the screen. Remember, each position on the screen consists of the ASCII character byte followed by the attribute byte. This is why BX is doubled.

To continue, the video page currently being used is loaded into BX, the row and column values in CX, and the length of the video page in AX. After the statements MUL BX and ADD AX,CX are executed, the direct video address we will write data to will be returned in AX—the offset address.

To turn the highlight bar off then, the code at the label OFF_NOW calls the GETPOS procedure to calculate the offset address and then loads the DL register with the screen's original attribute byte. The procedure BAR is then called to erase the highlight bar at the cursor's current position. Once this is done, ICU is finished.

The Highlight Bar

The procedure BAR is used to write new attributes to the screen (i.e., video memory). Usually the screen's data will use an attribute value of 07h (normal) and the attribute value of 70h (reverse) is used to show a bar across the screen. Note that color systems can use a value other than 70h to change the color of the highlight bar.

Therefore, the BAR procedure simply retrieves the number of columns displayed on the screen into CX and the row and column positions in BX and goes into a simple loop. This loop, in turn, calls the procedure WRCHAR. WRCHAR, shown below, stores the attribute in DL to video memory.

```
wrchar  proc    near
do_line:inc     bx                      ;move up to attribute byte
        mov     byte ptr[bx],dl         ;write new attribute to video
        inc     bx                      ;move up to character byte
        ret                             ;return to caller
wrchar  endp
```

The WRCHAR procedure only writes a single attribute byte to video memory at the offset address in BX. Since we want to change every character's attribute for the entire row, CX is set to the number of columns before the loop routine is entered. This makes WRCHAR execute for the total number of columns displayed on the screen.

Summary

ICU is by far the most complicated utility presented in this book, simply because it monitors the cursor's position through Int 10h, and it also monitors the keyboard through Int 09h. The discussion of how ICU was developed has shown you how to:

- change and modify interrupt service routines
- monitor the system to see if DOS is busy executing a service call
- shrink memory required by the TSR to a minimum in order to preserve the most memory possible for other TSR utilities and application programs

Projects

1. Add a command line switch that enables you to change the hotkey combination to another Alt or Ctrl keystroke combination. You should be able to change the hotkey even after the program is made memory resident.
2. If you have a color system, allow options for changing the color of the highlight bar.

For Further Study

If you want to learn more about using undocumented DOS function calls, consult Addition-Wesley's book *Undocumented DOS*.

An article published in *PC Magazine* (Volume 10, Number 3, February 12, 1991) also explains how to use several undocumented DOS services.

Program Listing

```
;ICU.ASM
;<c> 1992 by Deborah L. Cooper
;
codesg  segment                       ;start of code segment
        assume  cs:codesg,ds:codesg,es:codesg
        org     100h                    ;make this a COM program

start:  jmp     init                    ;skip past data area

copywr          db      'ICU Cursor Locating Utility'
                db      0dh,0ah
                db      '<c> 1992 by Deborah L. Cooper'
                db      '$'
adapter         db      2               ;0=MDA, 1=CGA, 2=EGA
attribute       db      70h             ;try 10h for color!
old_attrib      db      07h             ;normal attribute
page_size       dw      ?               ;video page size
video_segment   dw      0b800h          ;video segment address
cursor_pos      dw      ?               ;cursor position
addr_6845       dw      ?               ;CRT Controller base address
video_page      db      ?               ;current video page
orig_row        db      0               ;ROW position at time of install
bar_status      db      0               ;0=disable bar 1=enable bar
num_cols        dw      0               ;number of columns for this video
                                            ;mode
indos           dd      ?               ;INDOS flag address
res_segment     dw      ?               ;ICU's code segment

old10h          dd      ?               ;old Int 10h vector
old9h           dd      ?               ;old Int 09h vector

;==========================================================
;VIDEO interrupt handler for the screen. Every time an
;Int 10h interrupt occurs, execution comes here
;==========================================================
video   proc    near
        pushf                           ;needed to simulate Int
        call    cs:old10h               ;call for original routine
        cmp     cs:bar_status,1         ;do our CURSOR routine?
        jne     video_1                 ;no, skip it
        call    do_cursor               ;yes, do it now
video_1:iret                            ;all done, so exit
video   endp
;==========================================================
;KEYBD interrupt handler for the keyboard. Every time a
;key is pressed or released on the keyboard, execution comes
;here to our routine
;==========================================================
keybd   proc    far
        sti                             ;turn interrupts on
        push    ax                      ;save AX
        in      al,60h                  ;get scan code from keyboard
        cmp     al,34h                  ;our period keystroke?
        jne     kb2                     ;no, do normal keyboard then
        mov     ah,02h                  ;check shift key status function
        int     16h                     ;call bios
        cmp     al,8d                   ;is it Alt key too?
        jne     kb2                     ;no, do normal keyboard then
        call    kb_reset                ;yes, ignore the keystroke
        cmp     cs:bar_status,1       ;is bar toggled on?
        je      yes_bar               ;yes, go turn it off then
        mov     cs:bar_status,1       ;no, turn toggle switch to ON
        jmp     short kbl             ;and continue
yes_bar:mov     cs:bar_status,0       ;turn toggle switch to OFF
kbl:    call    do_cursor             ;do our routine
        pop     ax                    ;restore AX
        iret                          ;end of interrupt routine
kb2:    pop     ax                    ;recover AX
        jmp     cs:old9h              ;do original keyboard routine
keybd   endp
;=========================================================
;KB_RESET resets the keyboard and issues and EOI to the
;8259 PIC
;=========================================================
kb_reset        proc    near
        in      al,61h                ;get current control value
        mov     ah,al                 ;save in AH
        or      al,80h                ;set the high bit
        out     61h,al                ;send it to the control port
        mov     al,ah                 ;recover orginal value
        out     61h,al                ;send it out
        cli                           ;turn interrupts off
        mov     al,20h                ;load EOI value
        out     20h,al                ;send it to the 8259
        sti                           ;turn interrupts back on
        ret                           ;return to caller
kb_reset endp
;=========================================================
;DO_CURSOR is called by other routines to display the highlighted
;cursor bar across the screen
;=========================================================
do_cursor       proc    near
        sti                           ;turn interrupts on
        push    ax                    ;save all needed registers
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    ds
        push    es
        push    cs                    ;set DS to the code segment
        pop     ds
        assume  ds:codesg
        push    cs                    ;set ES to the code segment
        pop     es

;Make sure the current video mode is a text mode
        mov     ax,40h                ;point ES to the
        mov     es,ax                 ;Bios Data Area
        mov     al,es:[49h]           ;get video mode number in AL

        cmp     al,2                  ;is it mode 2?
        je      cur_1                   ;yes, proceed then
        cmp     al,3                    ;is it mode
        je      cur_1                   ;yes, proceed then
        cmp     al,7                    ;is it mode 7?
        je      cur_1                   ;yes, proceed then

exit:   pop     es                      ;restore registers and quit
        pop     ds
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret                             ;return to caller

;If DOS is busy (the INDOS flag) then just exit and try again later
cur_1:  push    di                      ;save DI
        push    es                      ;and ES
        les     di,cs:[indos]           ;exit if the INDOS
        cmp byte ptr es:[di],0          ;flag is non-zero
        jne     no_do                   ;exit NOW!
        pop     es                      ;restore ES
        pop     di                      ;and DI
        jmp     short cur_2             ;and do our routine
no_do:  pop     es                      ;restore ES
        pop     di                      ;and DI
        jmp     short exit              ;and quit altogether
;Save video parameters that must be used now or restored later
cur_2:  mov     al,es:[62h]             ;get current video page
        mov     video_page,al           ;save it
        mov     ax,es:[4ah]             ;get number of columns displayed
        mov     [num_cols],ax           ;for the video mode
        mov     ax,es:[4ch]             ;# of bytes for one video page
        mov     page_size,ax            ;(4,000 for 80 x 25 mode) etc
        mov     ax,es:[50h]             ;get cursor address from CRTC
        mov     cursor_pos,ax           ;save it
        cmp     bar_status,0            ;request to toggle it OFF?
        je      off_now                 ;yes, go turn highlight off
        xchg    al,ah                   ;AL=row AH=column
;First we must turn the currently highlighted bar OFF
ll_fix: mov     al,orig_row             ;get previous row position
        xor     ah,ah                   ;set column position to zero
        mov     bl,video_page           ;get video page in BL
        call    getpos                  ;DI=direct video address
        mov     dl,old_attrib           ;original attribute in AL
        call    bar                     ;erase the bar
;Now turn the highlight bar on at the current cursor position
        mov     ax,cursor_pos           ;get the app's cursor position
        xchg    ah,al                   ;AL=row AH=column
        xor     ah,ah                   ;set column position to zero
        xor     bh,bh                   ;zero top half first
        mov     bl,video_page           ;video page in BL
        call    getpos                  ;DI=direct video address
        mov     dl,attribute            ;set AL to new attribute
        call    bar                     ;display the bar now
        mov     ax,cursor_pos           ;get cursor position again
        mov     orig_row,ah             ;save row value for next time
        jmp     exit                    ;and quit
off_now:xor     ah,ah                   ;set column to zero
        mov     al,orig_row             ;get row in AL now
        call    getpos                  ;calculate video address in DI
        mov     dl,old_attrib           ;set AL to original attribute
        call    bar                     ;turn the highlighted bar off!
        jmp     exit                    ;and exit
do_cursor       endp
;==========================================================
;GETPOS calculates the offset address in video memory that
;corresponds to the current row, column and video page of
;the current cursor location
;On entry:  AH,AL = row and column,  BX = video page
;On exit:   AX = offset address
;==========================================================
getpos  proc    near
        mov     bx,ds:[num_cols]        ;we must multiply the row
        shl     bx,1                    ;value by # bytes per video row
        mul     bl                      ;multiply columns by 2
        xor     bh,bh                   ;convert DX to a word now
        mov     bl,video_page           ;add results
        push    cx
        mov     cx,ax                   ;row and column in CX
        mov     ax,page_size            ;length of a video page
        mul     bx                      ;multiply page length by # pages
        add     ax,cx                   ;AX=video address
        pop     cx
        ret                             ;return to caller
getpos  endp
;==========================================================
;WRCHAR writes the character and attribute for the current
;row and column and video page as specified
;On entry:  ES:BX = video address to write to
;           DL = attribute to write
;==========================================================
wrchar  proc    near
do_line:inc     bx                      ;move up to attribute byte
        mov     byte ptr[bx],dl         ;write new attribute to video
        inc     bx                      ;move up to character byte
        ret                             ;return to caller
wrchar  endp
;==========================================================
;BAR reads/writes each character/attribute pair from/to the
;screen and changes the attribute for the entire row
;On entry:  DL = attribute to write/read
;==========================================================
bar     proc    near
        mov     cx,num_cols             ;maximum 80 attributes to modify
        mov     bx,ax                   ;row and column in BX
        push    ds                      ;save DS first
        mov     ds,video_segment      ;set DS:SI to video segment
bar_1:  call    wrchar                ;write new attribute to screen
        loop    bar_1                 ;go do next column
        pop     ds                    ;restore DS
        ret                           ;return to caller
bar     endp

end_prog        =       $             ;marks end of resident code
;==========================================================
;INIT makes the program resident in memory
;==========================================================
already_in$     db      'ICU is already resident!','$'
out_now         db      'ICU was removed from memory','$'
no_way          db      'Unable to remove ICU from memory!','$'

init    proc    near
        assume  cs:codesg,ds:codesg
        cld                           ;clear DF first
        not     word ptr start        ;destroy this first data
        xor     bx,bx                 ;search from first segment
        mov     ax,cs                 ;compare to this code segment
next:   inc     bx                    ;look at next segment
        cmp     ax,bx                 ;until reaching this code segment
        mov     es,bx
        je      not_in                ;not installed yet
        mov     si,offset start       ;setup to compare strings
        mov     di,si
        mov     cx,16                 ;compare 16 bytes
        rep cmpsb                     ;do it now
        or      cx,cx                 ;did strings match?
        jnz     next                  ;no, try next segment
        mov     res_segment,es        ;save ICU's code segment
        jmp     not_in                ;go check for commands

;Now check the command line for possible switches
not_in: mov     si,80h                ;address of command line
        lodsb                         ;get length in AL
        or      al,al                 ;are there any switches?
        jz      make_r                ;no, go install then
com_2:  lodsb                         ;get next byte from line
        cmp     al,0dh                ;is it a carriage return?
        je      make_r                ;yes, exit this loop then!
        cmp     al,20h                ;is it a space?
        je      com_2                 ;yes, skip leading spaces
        cmp     al,'/'                ;is it a slashbar?
        jne     make_r                ;no, go install then
        lodsb                         ;yes, get next byte
        and     al,5fh                ;convert to uppercase
        cmp     al,'U'                ;want to uninstall?
        jne     make_r                ;no, go install it
        jmp     uninstall             ;yes, remove it now

;Determine what type of video adapter is installed in the system
make_r: mov     ax,40h                ;AX = BIOS segment
```

```
        mov     es,ax                   ;into ES
        mov     al,es:[49h]             ;get video mode
        cmp     al,7                    ;is it mode 7?
        jne     init_3                  ;no, then its a color system

;Modify video parameter values for monochrome
init_2: sub     video_segment,800h      ;set for monochrome
init_3: mov     bar_status,1            ;toggle bar to ON to start with

;Determine the port address of the CRT Controller and store it
        mov     ax, es:[50h]            ;get cursor address from BIOS
        mov     orig_row,ah             ;save the ROW value for later

;Save the INDOS flag address
        mov     ah,34h                  ;get address of
        int     21h                     ;the INDOS flag
        mov     word ptr indos,bx       ;save offset address
        mov     word ptr indos[2],es    ;save segment address

;Save and replace interrupt vectors
        mov     ah,35h                  ;get interrupt 10h vector
        mov     al,10h                  ;for BIOS and timer
        int     21h                     ;call dos
        mov     word ptr old10h,bx      ;save offset address
        mov     word ptr old10h[2],es   ;save segment address

        mov     ah,25h                  ;set interrupt vector function
        mov     dx,offset video         ;to this routine
        int     21h                     ;call dos

        mov     ah,35h                  ;get interrupt vector function
        mov     al,09h                  ;for the keyboard
        int     21h                     ;call dos
        mov     word ptr old9h,bx       ;save offset address
        mov     word ptr old9h[2],es    ;save segment address

        mov     ah,25h                  ;set interrupt vector function
        mov     dx,offset keybd         ;to this routine
        int     21h                     ;call dos

;Display the copyright notice
        mov     dx,offset copywr        ;copyright notice
        mov     ah,09h                  ;display string function
        int     21h                     ;call dos
;Release environment block (to save some memory!)
        mov     ax,ds:[002ch]
        mov     es,ax
        mov     ah,49h                  ;release memory block function
        int     21h                     ;call dos

;Terminate but remain in memory
        mov     dx,(offset end_prog-offset codesg + 15) shr 4
        mov     ax,3100h                ;terminate function
        int     21h                     ;call dos
```

```
;Attempt to remove ICU from memory
uninstall:
        assume  ds:codesg
        push    es                    ;save ES first
        mov     ah,35h                ;and see if ICU still
        mov     al,10h                ;BIOS/timer interrupt
        int     21h                   ;owns it
        mov     ax,es                 ;compare vector address
        cmp     ax,res_segment        ;with ICU's code segment
        jne     un_1                  ;go if cannot uninstall
;Release the memory occupied by ICU now
        mov     ah,49h                ;free memory function
        int     21h                   ;call dos
        jc      un_1                  ;go if error
restore:push    ds
        assume  ds:nothing
        lds     dx,es:[old10h]        ;DX=video interrupt address
        mov     ah,25h                ;set interrupt vector address
        mov     al,10h                ;for Int 10h
        int     21h                   ;call dos
        lds     dx,es:[old9h]         ;DX=keyboard interrupt address
        mov     ah,25h                ;set interrupt vector address
        mov     al,09h                ;for the keyboard
        int     21h                   ;call dos
        pop     ds
        assume  ds:codesg
        not word ptr es:[start]       ;remove fingerprint
        push    cs                    ;set DS to our
        pop     ds                    ;code segment again
        assume  ds:codesg
        mov     dx,offset out_now     ;ICU removed message
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        pop     es                    ;recover ES too
        mov     ah,4ch                ;terminate program function
        int     21h                   ;call dos
un_1:   mov     dx,offset no_way      ;cannot uninstall ICU!
        mov     ah,09h                ;display string function
        int     21h                   ;call dos
        pop     es                    ;restore ES now
        mov     ah,4ch                ;terminate program function
        int     21h                   ;call dos
init    endp
codesg  ends                          ;end of code segment
        end     start                 ;end of program
```

Appendix A

The Program Segment Prefix

The Program Segment Prefix (PSP) is created by DOS each time a program is loaded into memory. The PSP is 256 bytes in length. Each field in the PSP describes some aspect of the currently running program. Programmers can read this information at any time. It is, however, highly recommended that the information stored in the PSP not be modified or unpredictable results could occur.

Program Segment Prefix (PSP) Structure

Offset	*Length*	*Description*
00h	2	Int 20h (terminate)
02h	2	Segment address of the top of the current program's memory allocation block
04h	1	Reserved by DOS
05h	1	Int 21h instruction
06h	2	Available memory—the number of bytes available in the segment
08h	2	Reserved
0Ah	4	Terminate address (IP, CS). Address of the termination interrupt handler (Int 22h) inherited by the current program. DOS uses this value to restore the Int 22h vector when the program terminates
0Eh	4	Ctrl+Break address (IP, CS). Address of the Ctrl+Break interrupt handler (Int 23h) inherited by the current program. DOS uses this value to restore the Int 23h vector when the program terminates

12h	4	Critical error address (IP, CS). Address for the Critical Error interrupt handler (int 24h) inherited by the current program. DOS uses this value to restore the Int 24h vector when the program terminates
16h	22	Parent process's Program Segment Prefix
2Ch	2	Segment address of the DOS environment
2Eh	46	Reserved by DOS
5Ch	16	First File Control Block, as parsed from the first parameter in the command tail
6Ch	20	Second File Control Block, as parsed form the second parameter in the command tail
80h	1	Length, in bytes, of the command tail
81h	127	Command tail. Also used as default DTA for FCB functions

Appendix B

IBM PC Character Set

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00		24	18	↑	48	30	0
1	01	☺	25	19	↓	49	31	1
2	02	☻	26	1A	→	50	32	2
3	03	♥	27	1B	←	51	33	3
4	04	♦	28	1C	∟	52	34	4
5	05	♣	29	1D	↔	53	35	5
6	06	♠	30	1E	▲	54	36	6
7	07	•	31	1F	▼	55	37	7
8	08	◘	32	20		56	38	8
9	09	○	33	21	!	57	39	9
10	0A	◙	34	22	"	58	3A	:
11	0B	♂	35	23	#	59	3B	;
12	0C	♀	36	24	$	60	3C	<
13	0D	♪	37	25	%	61	3D	=
14	0E	♫	38	26	&	62	3E	>
15	0F	☼	39	27	’	63	3F	?
16	10	►	40	28	(	64	40	@
17	11	◄	41	29	)	65	41	A
18	12	↕	42	2A	*	66	42	B
19	13	‼	43	2B	+	67	43	C
20	14	¶	44	2C	,	68	44	D
21	15	§	45	2D	-	69	45	E
22	16	▬	46	2E	.	70	46	F
23	17	↨	47	2F	/	71	47	G

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
72	48	H	105	69	i	138	8A	è
73	49	I	106	6A	j	139	8B	ï
74	4A	J	107	6B	k	140	8C	î
75	4B	K	108	6C	l	141	8D	ì
76	4C	L	109	6D	m	142	8E	Ä
77	4D	M	110	6E	n	143	8F	Å
78	4E	N	111	6F	o	144	90	É
79	4F	O	112	70	p	145	91	æ
80	50	P	113	71	q	146	92	Æ
81	51	Q	114	72	r	147	93	ô
82	52	R	115	73	s	148	94	ö
83	53	S	116	74	t	149	95	ò
84	54	T	117	75	u	150	96	û
85	55	U	118	76	v	151	97	ù
86	56	V	119	77	w	152	98	ÿ
87	57	W	120	78	x	153	99	Ö
88	58	X	121	79	y	154	9A	Ü
89	59	Y	122	7A	z	155	9B	¢
90	5A	Z	123	7B	{	156	9C	£
91	5B	[	124	7C	|	157	9D	¥
92	5C	\	125	7D	}	158	9E	₧
93	5D	]	126	7E	~	159	9F	ƒ
94	5E	^	127	7F	⌂	160	A0	á
95	5F	_	128	80	Ç	161	A1	í
96	60	‘	129	81	ü	162	A2	ó
97	61	a	130	82	é	163	A3	ú
98	62	b	131	83	â	164	A4	ñ
99	63	c	132	84	ä	165	A5	Ñ
100	64	d	133	85	à	166	A6	ª
101	65	e	134	86	å	167	A7	º
102	66	f	135	87	ç	168	A8	¿
103	67	g	136	88	ê	169	A9	⌐
104	68	h	137	89	ë	170	AA	¬

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
171	AB	½	200	C8	╚	229	E5	σ
172	AC	¼	201	C9	╔	230	E6	µ
173	AD	¡	202	CA	╩	231	E7	τ
174	AE	«	203	CB	╦	232	E8	Φ
175	AF	»	204	CC	╠	233	E9	Θ
176	B0	░	205	CD	═	234	EA	Ω
177	B1	▒	206	CE	╬	235	EB	δ
178	B2	▓	207	CF	╧	236	EC	∞
179	B3	│	208	D0	╨	237	ED	ø
180	B4	┤	209	D1	╤	238	EE	ε
181	B5	╡	210	D2	╥	239	EF	∩
182	B6	╢	211	D3	╙	240	F0	≡
183	B7	╖	211	D4	╘	241	F1	±
184	B8	╕	213	D5	╒	242	F2	≥
185	B9	╣	214	D6	╓	243	F3	≤
186	BA	║	215	D7	╫	244	F4	⌠
187	BB	╗	216	D8	╪	245	F5	⌡
188	BC	╝	217	D9	┘	246	F6	÷
189	BD	╜	218	DA	┌	247	F7	≈
190	BE	╛	219	DB	█	248	F8	°
191	BF	┐	220	DC	▄	249	F9	∙
192	C0	└	221	DD	▌	250	FA	·
193	C1	┴	222	DE	▐	251	FB	√
194	C2	┬	223	DF	▀	252	FC	ⁿ
195	C3	├	224	E0	α	253	FD	²
196	C4	─	225	E1	ß	254	FE	■
197	C5	┼	226	E2	Γ	255	FF	
198	C6	╞	227	E3	π			
199	C7	╟	228	E4	Σ			

Appendix C

DOS and BIOS Functions Used in This Book

Int 10h, Function 02h
Set Cursor Position

This function sets the cursor to a specific row and column position on the screen.

To call: AH = 02h
BH = Video page
DH = Row position
DL = Column position

Returns: Nothing

Comments:

```
mov  ah, 02h
mov  bh, φ
mov  dh, φ
mov  dl, φ
int  10 h
```

In graphic mode, the video page specified in BH should be set to 0. In text mode, the row and column positions are specified for the upper left corner as 0,0 and the lower right corner as 79,24 in standard 80 x 25 text mode.

Int 10h, Function 0Eh
Write Character

This function outputs a character to the screen.

To call: AH = 0Eh
AL = ASCII character code
BH = Video page (text mode)
BL = Foreground color (graphics mode)

Returns: Nothing

```
mov  ah, 0eh
mov  al, a
mov  bh, φ
  or
mov  bl, φ
```

Black foreground color

Comments:

This function provides some character processing services for the bell (07h), backspace (08h), carriage return (0Dh), and linefeed (0Ah) characters. After this service is executed, the cursor is advanced to the next character position.

Int 16h, Function 00h
Get Keyboard Character

This function waits until a character is available and reads that character from the keyboard buffer.

To call:	AH = 00h

Returns:	AH = Keyboard scan code
	AL = ASCII character code

Comments:	None

Int 16h, Function 02h
Get Keyboard Flags

This function retrieves the status of the shift and toggle keys from the BIOS Data Area.

To call:	AH = 02h
Returns:	AL = Keyboard status as shown below

Bits 76543210	*Meaning*
00000001	Right shift depressed
00000010	Left shift depressed
00000100	Ctrl key depressed
00001000	Alt key depressed
00010000	ScrollLock enabled
00100000	NumLock enabled
01000000	CapsLock enabled
10000000	Insert enabled

Comments: None

Int 17h, Function 02h
Get Printer Status

This function retrieves the current status of the printer.

To call: AH = 02h
DX = Printer number (0, 1, or 2)

Returns: AH = Printer status as shown below

Bits 76543210	*Meaning*
00000001	Time out
00000110	Not used
00001000	I/O error
00010000	Printer selected
00100000	Out of paper
01000000	Acknowledged
10000000	Printer not busy

Int 21h, Function 02h
Display Character

This function displays a character on the standard output device (usually the screen).

To call: AH = 02h
DL = Character to display

Returns: Nothing

Comments:

Function 02h, when asked to display a backspace (08h) character, will move the cursor back one position on the screen, but the character is not erased (nondestructive backspace).

Int 21h, Function 09h
Display String

Sends a string of ASCII characters to standard output (usually the screen). The string must be terminated with a dollar sign ($) byte.

To call: AH = 09h
DS:DX = Pointer to string

Returns:	Nothing

Comments:

The terminating dollar sign is not sent to the output device. In addition, the dollar sign cannot be embedded within the string.

Int 21h, Function 0Ah
Buffered Keyboard Input

This function reads a string of characters from standard input (usually the keyboard) and echoes it to standard output (usually the screen). The function is terminated when it receives a carriage return (0Dh) byte or when the maximum length for the string has been reached.

To call:	AH = 0Ah DS:DX = Pointer to input buffer
Returns:	Nothing

Comments:

The input buffer must be in the format:

Offset	*Description*
00h	The maximum length for the string, not exceeding 255 characters, including the carriage return byte. This value is set by your program, not DOS.
01h	The actual number of characters contained in the string, excluding the carriage return byte. This value is set by Function 0Ah itself.
02h+	The actual characters received.

Int 21h, Function 0Eh
Set Current Drive

This function sets the specified drive to be the new default drive.

To call:	AH = 0Eh
	DL = Drive number (0=A, 1=B, etc.)

Returns:	AL = Last drive number

Comments:

This function also returns a count of the number of logical disk drives (RAM disks, floppy disks and logical disks) installed in the computer system.

Int 21h, Function 19h
Get Default Drive

This function retrieves the number of the current default disk drive.

To call: AH = 19h

Returns: AL = Drive number (0=A, 1=B, etc.)

Comments: None

Int 21h, Function 1Ah
Set Disk Transfer Address

This function sets the address of the Disk Transfer Address (DTA) to a specified buffer that DOS uses for file I/O functions.

To call: AH = 1Ah
DS:DX = Pointer to new DTA buffer

Returns: Nothing

Comments:

When DOS executes a program, the default DTA is set at offset 80h within the Program Segment Prefix (PSP) and is 128 bytes in length. The new DTA buffer must be large enough to accommodate the largest block of data to be manipulated by file I/O functions.

Int 21h, Function 25h
Set Interrupt Vector

This function substitutes the current interrupt vector routine with a user-written routine.

To call: AH = 35h
AL = Interrupt number
DX:DX = Address of new interrupt handler

Returns: Nothing

Comments:

This function is used primarily in terminate and stay resident (TSR) programs to replace a specific interrupt vector. The interrupt number must be in the range 00h-FFh. Note that Function 35h of Int 21h should be used to first retrieve and save the address of the original interrupt handler. Function 25h can then be used to restore the original handler at a later time in the program.

Int 21h, Function 30h
Get DOS Version

This function retrieves the version number (major and minor) for DOS installed in the computer system.

To call:	AH = 30h

Returns:	AL = Major version number
	AH = Minor version number
	BX = 00h

Comments:

In DOS 5.0, this function returns the OEM number or the version flag in BH and a 24-bit serial number in BL:CX. For earlier DOS versions, BX and CX are set to zero. Function 30h returns the DOS version number as set by the SETVER command.

Int 21h, Function 31h
Terminate and Stay Resident

This function terminates a program, leaving a portion of itself resident in memory.

To call:	AH = 31h
	AL = Return code
	DX = Number of paragraphs of memory to reserve

Returns:	Nothing

Comments:

This function is used in terminate and stay resident (TSR) programs. A return code may be used which can be tested by a parent program (Function 4Dh of Int 21) or a batch file (IF ERRORLEVEL) to indicate special exit conditions.

Int 21h, Function 34h
Get Address of INDOS Flag

This function retrieves the address of the INDOS Flag.

To call:	AH = 34h
Returns:	ES:BX = Segment:offset address of the flag

Comments:

This function is used primarily in terminate and stay resident (TSR) programs. If DOS is currently processing an Int 21h service, the INDOS Flag's value is non-zero.

Int 21h, Function 35h
Get Interrupt Vector

This function retrieves the segment and offset address of the current interrupt handler routine for the specified interrupt number.

To call:	AH = 35h
	AL = Interrupt number

Returns:	ES:BX = Segment:offset address of interrupt handler

Comments:

This function is used primarily in terminate and stay resident (TSR) programs to retrieve the segment and offset address of the routine that processes the specified interrupt. The interrupt number must be in the range 00h-FFh.

Int 21h, Function 39h
Create Directory

This function creates a new subdirectory.

To call:	AH = 39h DS:DX = Address of ASCIIZ pathname	
Returns:	Carry Flag clear if successful Carry Flag set if not successful	
	AX=03h	Path not found
	AX=05h	Access denied

Comments:

Function 39h will create a new directory on the specified disk. If the directory already exists, an error will occur. In addition, if the root directory is full, an error will be generated.

Int 21h, Function 3Bh
Set Current Directory

This function allows you to change the current directory to a new default directory.

To call: AH = 3Bh
DS:DX = Pointer to new pathname

Returns: Carry Flag clear if successful
Carry Flag set if not successful
AX=03h Path not found

Comments:

The directory name passed in DS:DX must be in ASCIIZ format. This name may contain a drive designator, in which case the directory will be changed on the specified disk drive. However, the default disk will not be changed.

Int 21h, Function 3Ch
Create File

Creates and opens a new file. If the file already exists, its length is truncated to zero.

To call: AH = 3Ch
CX = File attribute
DS:DX = Pointer to file specification

Returns: Carry Flag clear if successful
AX = File handle
Carry Flag set if not successful
AX=03h Path not found
AX=04h No file handles available
AX=05h Access denied

Comments:

The file specification must be in the format of an ASCIIZ string. This string may contain a drive and path designation. The newly created file is assigned the first available file handle by DOS. The file attribute specified in CX may be set to any combination of the following:

Value	*Description*
00h	Normal file. Data can be read from or written to the file.
01h	Read only file. Data can be read from the file only.
02h	Hidden file. The file is hidden; it cannot be seen in the directory listing.
04h	System file.
08h	Volume label file.
20h	Archive file. The file is marked for archive or backup purposes.

When creating a Volume Label file, only one such file may exist on each disk drive, providing that another Volume Label file for the specified drive does not already exist.

Int 21h, Function 3Eh
Close File

This function closes a file that was previously opened or created with a file-handle function.

To call: AH = 3Eh
BX = File handle

Returns: Carry Flag clear if successful
Carry Flag set if not successful
AX=06h Invalid file handle

Comments:

When the file is closed, all internal buffers for the file are flushed (i.e., any pending write functions are completed). The directory entry for the file is also updated (if the file was modified) with the new file size, date and time. In addition, the file handle is released back to the operating system for use by another application program.

Int 21h, Function 3Fh
Read from File or Device

This function reads a specified number of bytes from a file, placing the data in the designated I/O buffer.

To call: AH = 3Fh
BX = File handle

CX = Number of bytes to read
DS:DX = Segment:offset address of I/O buffer

Returns: Carry Flag clear if successful
AX=Number of bytes actually read from file
Carry Flag set if not successful

AX=05h	Access denied
AX=06h	Invalid file handle

Comments:

If the end of the file is reached, AX will contain a count of the actual number of bytes read from the file or device. If this value is less than the count in CX, a partial record will have been read. In the event that Function 3Fh returns a 0 value in AX, the file pointer is at the end of the file.

Int 21h, Function 40h
Write to File or Device

This function writes a specified number of bytes from an I/O buffer to a file or device.

To call: AH=40h
BX=File handle
CX=Number of bytes to write
DS:DX = Segment:offset address of I/O buffer

Returns: Carry Flag clear if successful
AX=Number of bytes written to file
Carry Flag set if not successful

AX=05h	Access denied
AX=06h	Invalid file handle

Comments:

When this function has been executed, the values in CX and AX are usually identical. If AX is less than CX, then a partial record was written to the file or device.

Int 21h, Function 47h
Get Current Directory

This function returns an ASCIIZ string containing the name of the current directory.

To call: AH = 47h
DL = Drive code (0=Default, 1=A, etc.)
DS:DI = Segment:offset address of directory pathname

Returns: Carry Flag clear if successful
DS:SI = Segment:offset address of directory name
Carry Flag set if not successful
AX=0Fh Invalid drive code

Comments:

On return from this function, the ASCIIZ pathname stored in the buffer pointed to by DS:SI does not contain a drive specifier or a leading backslash (\) character. If the directory is the root directory, the first byte of the buffer will be a NULL (0) byte.

Int 21h, Function 49h
Release Block of Memory

This function releases a block of memory back to the system pool that was previously allocated by Function 48h of Int 21h, Allocate Memory.

To call: AH = 49h
ES = Segment address of memory block to be released

Returns: Carry Flag clear if successful
Carry Flag set if not successful
AX=07h Memory control blocks destroyed
AX=09h Invalid memory block address

Comments:

The block of memory this function attempts to release must have previously been allocated by Function 48h or unpredictable results could occur.

Int 21h, Function 4Ch
Terminate Process with Return Code

This function is used to terminate a program, returning control to the operating system.

To call: AH = 4Ch
AL = Return code

Returns: Nothing

Comments:

The value in AL is set by a program to indicate special exit conditions, which may be evaluated by a parent program or by a batch file.

Int 21h, Function 4Eh
Find First Matching File

This function searches the disk for the first occurrence of a matching file specification.

To call: AH = 4Eh
CX = File attribute
DS:DX = Segment:offset address of ASCIIZ file specification

Returns: Carry Flag clear if successful
Carry Flag set if not successful

AX=02h	File not found
AX=03h	Invalid path
AX=12h	No more matching files

Comments:

The ASCIIZ file specification passed in DS:DX may contain a drive and path specifier, as well as wildcard characters in the filename itself. Function 4Eh places information about the found file in the current default DTA. The contents of the DTA, which is 43 bytes long, is as follows:

Offset	*Length*	*Description*
00h	21 bytes	Reserved by DOS
15h	1 byte	File attribute
16h	1 word	File time
18h	1 word	File date

1Ah	Doubleword file size	
1Eh	13 bytes	ASCIIZ filename and extension

Note that this function will only return information for files that match the attribute(s) specified in CX. The possible file attributes are as follows:

Value	*Description*
00h	Normal
02h	Normal and hidden
04h	Normal and system
06h	Normal, system, and hidden
08h	Volume label
10h	Directory

Int 21h, Function 4Fh
Find Next Matching File

This function searches the disk for the next occurrence of a matching file specification.

To call: AH = 4Fh

Returns: Carry Flag clear if successful
Carry Flag set if not successful

AX=02h	File not found
AX=03h	Invalid path
AX=12h	No more matching files

Comments:

This function uses the file specification and file attribute used by Function 4Eh, Find First Matching File. Therefore, a call must have been previously made to Function 4Eh before Function 4Fh can be invoked. The information returned is the same as for Function 4Eh and is placed in the DTA buffer for each matching file the function finds.

Int 21h, Function 56h
Rename or Move File

This function renames a file or moves a file to another directory on the same disk drive.

To call: AH = 56h
DS:DX = Segment:offset address of old file
ES:DI = Segment:offset address of new file

Returns: Carry Flag clear if successful
Carry Flag set if not successful

AX=02h	File not found
AX=03h	Path not found
AX=05h	Access denied
AX=11h	Not same device

Comments:

The names of the files addressed by DS:DX and ES:DI must be in ASCIIZ format. The file specifications can include a drive specifier and path, but wildcard characters are not supported. Subdirectories can be renamed if the DOS version is 3.0 or greater. Files can only be moved on the specified disk drive; in other words, Function 56h cannot be used to move a file from one disk to another.

Int 2Fh, Function AE00h
Check for Installed DOS Command

This function determines whether the specified command is an internal or external TSR extension to COMMAND.COM. It enables non pop-up TSR programs to become part of the operating system.

To call: AH = AEh
AL = 00h
DS = FFFFh
DS:BX = Segment:offset address of command line

Returns:

AL = FFh	This command is a TSR extension to COMMAND.COM
AL = 00h	This command should be executed in the normal fashion

Comments:

DS:BX must point to the address of the command line. This command line must be in the format:

1 byte	Maximum length of command line
1 byte	A count of the number of bytes to follow
n bytes	The command itself, terminated by a carriage return (0Dh) byte

Int 2Fh, Function AE01h
Execute Installed Command

This function executes an installed non pop-up TSR program, whether its a DOS internal command or not.

To call: AH = AEh
AL = 01h
DS:SI = Segment:offset address of buffer

Returns: The function returns "buffer" filled with a length byte, followed by the internal command to execute. The command is in uppercase. If the length of the command is 0, the command will not be executed.

Comments:

Sub-function 00h of AEh must be called first and must indicate that the command requested is resident in memory before issuing this service call.

Bibliography

Abrash, Michael. *Zen of Assembly Language: Volume 1, Knowledge.* Glenview, IL: Scott, Foresman and Company, 1990.

Dettman, Terry. *DOS Programmer's Reference.* Indianapolis: QUE Corporation, 1989. Second edition, revised by Jim Kyle.

Duncan, Ray. *IBM ROM BIOS.* Redmond, WA: Microsoft Press, 1988.

__________ *MSDOS Functions.* Redmond, WA: Microsoft Press, 1988.

Microsoft Press. *Microsoft MSDOS Programmer's Reference.* Redmond, WA: Microsoft Press, 1991.

Mueller, John and Wang, Wallace. *The Ultimate DOS Programmer's Manual.* Blue Ridge Summit, PA: Windcrest Books, 1991.

Prosise, Jeff. *PC Magazine DOS 5 Techniques and Utilities.* Emeryville, CA: Ziff-Davis Press, 1991.

Schulman, Andrew...[et al.]. *Undocumented DOS: A Programmer's Guide to Reserved MSDOS Functions and Data Structures.* Reading, MA: Addison Wesley, 1990.

Swan, Tom. *Mastering Turbo Assembler.* Indianapolis, IN: Hayden Books, 1989.

Tischer, Michael. *PC Intern System Programming: The Encyclopedia of DOS Programming Know How.* Grand Rapids, MI: Abacus, 1992.

Williams, Al. *DOS 5: A Developer's Guide: Advanced Programming Guide to DOS.* Redwood City, CA: M & T Books, 1991.

```
mov    dx, offset □ one   }
mov    ah, 09h            }  Display Hello
int    21h                }
```

```
inkey: mov    ah, ØØh     ; get a keystroke
       int    16 h        ; call Bios
       cmp    al, 'Y'     ; the char is in al
       je     exit
       jmp    inkey       ;
```

<u>output one char 553 Jamsa</u>

```
mov □ al, 41h    ; A is = ascii 41 H
mov   ah, 9h     ; Request a video write
mov   bx, 7h     ; character attribute,
mov   cx, 1h     ; number of chars to write
int   10         ; perform the video svc
```

Index

24 write one char to screen Pom

Popular Applications Series

Developing Utilities in Assembly Language

Written with the programmer in mind, this book focuses on programming techniques readers can use when creating their own programs. Each section discusses the development and theories used to create working programs and includes a general program description, a list of DOS and BIOS function calls, development explanations, summary descriptions, and the actual ASM source code listing for the program. A companion diskette with debugged source code is included.

DEBORAH L. COOPER is a professional programmer and writer with an extensive background in the computer industry. A power user of a variety of software applications, Ms. Cooper's published works include software manuals and numerous newspaper and magazine articles.

Book Buyer's Guide	
User Category	☑ New □ Experienced □ Advanced ☑ Professional ☑ Educator
Book Type:	☑ Tutorial -- For: ☑ Self Teaching ☑ Classroom Use □ Command Reference □ Advanced Topics ☑ Working Examples -- ☑ Source Code □ Templates ☑ Hands-on Activities □ Other ☑ Companion Diskette --- ☑ Included □ Available

51595

9 781556 224478

Cover Design by Alan McCuller

WORDWARE PUBLISHING, INC.
—First in Quality—

ISBN 1-55622-429-X

$15.95